Contents

- Debugging Tools for Windows (WinDbg, KD, CDB, NTSD)
- Download Debugging Tools for Windows - WinDbg
  - Download Debugging Tools for Windows - WinDbg
  - Download Windows Symbol Packages for Debugging
- Getting Started with Windows Debugging
  - Getting Started with Windows Debugging
  - Getting Started with WinDbg (User-Mode)
  - Getting Started with WinDbg (Kernel-Mode)
  - Choosing the 32-Bit or 64-Bit Debugging Tools
  - Debugging Environments
  - Setting Up Debugging (Kernel-Mode and User-Mode)
    - Setting Up Debugging (Kernel-Mode and User-Mode)
    - Setting Up Kernel-Mode Debugging
      - Setting Up Kernel-Mode Debugging
      - Setting Up KDNET Network Kernel Debugging Automatically
      - Setting Up KDNET Network Kernel Debugging Manually
      - Setting Up Kernel-Mode Debugging over a 1394 Cable Manually
      - Setting Up Kernel-Mode Debugging over a USB 3.0 Cable Manually
      - Setting Up Kernel-Mode Debugging over a USB 2.0 Cable Manually
      - Setting Up Kernel-Mode Debugging over a Serial Cable Manually
      - Setting Up Kernel-Mode Debugging using Serial over USB Manually
      - Setting Up Kernel-Mode Debugging of a Virtual Machine Manually using a COM Port
      - Setting Up Network Debugging of a Virtual Machine Host - KDNET
      - Setting Up Local Kernel Debugging of a Single Computer Manually
    - Supported Ethernet NICs for Network Kernel Debugging in Windows 10
    - Supported Ethernet NICs for Network Kernel Debugging in Windows 8.1
    - Supported Ethernet NICs for Network Kernel Debugging in Windows 8
    - Configuring tools.ini

- Using KDbgCtrl
- Debug Universal Drivers - Step by Step Lab (Echo Kernel-Mode)
- Debug Drivers - Step by Step Lab (Sysvad Kernel-Mode)

Debugging Using WinDbg Preview

- Debugging Using WinDbg Preview
- WinDbg Preview - What's New
- WinDbg Preview - Installation
- WinDbg Preview - Command line startup options
- WinDbg Preview – Settings and workspaces
- WinDbg Preview – Keyboard shortcuts
- WinDbg Preview - Start a user-mode session
- WinDbg Preview - Start a kernel mode session
- WinDbg Preview – File menu
- WinDbg Preview – Home menu
- WinDbg Preview – View menu
- WinDbg Preview – Breakpoints
- WinDbg Preview – Data Model
- WinDbg Preview – Scripting
- Time Travel Debugging - Overview
  - Time Travel Debugging - Overview
  - Time Travel Debugging - Recording
  - Time Travel Debugging - Replay a trace
  - Time Travel Debugging - Working with trace files
  - Time Travel Debugging - Troubleshooting
  - Time Travel Debugging - Navigation commands
  - Time Travel Debugging - Extension Commands
    - Time Travel Debugging - Extension Commands
    - Time Travel Debugging - !tt (time travel)
    - Time Travel Debugging - !index (time travel)
    - Time Travel Debugging - !positions (time travel)
  - Time Travel Debugging - Sample App Walkthrough
  - Time Travel Debugging - Introduction to Time Travel Debugging objects

Time Travel Debugging - Introduction to Time Travel Debugging objects
Time Travel Debugging - Calls objects
Time Travel Debugging - Heap objects
Time Travel Debugging - Memory objects
Time Travel Debugging - Event objects
Time Travel Debugging - Exception objects
Time Travel Debugging - Position objects
Time Travel Debugging - Range objects
Time Travel Debugging - Thread objects
Time Travel Debugging - JavaScript Automation
Debugging Resources
Debugging Resources
Debugging Tools for Windows: New for Windows 10
Tools Included in Debugging Tools for Windows
Tools Included in Debugging Tools for Windows
ADPlus
DumpChk
GFlags
GFlags
GFlags Overview
GFlags Details
GFlags Commands
GFlags Flag Table
GFlags and PageHeap
Monitoring Silent Process Exit
Global Flags Dialog Box
Global Flags Dialog Box
Opening the Dialog Box
Setting and Clearing System-wide Flags
Setting and Clearing Kernel Flags
Setting and Clearing Image File Flags
Configuring Silent Process Exit Monitoring

- Launching a Program with Flags
- Running a Program in a Debugger
- Configuring Special Pool
  - Configuring Special Pool
  - Requesting Special Pool by Pool Tag
  - Requesting Special Pool by Allocation Size
  - Canceling Requests for Special Pool
  - Detecting Overruns and Underruns
- Configuring Object Reference Tracing

Global Flag Reference

- Global Flag Reference
- Buffer DbgPrint Output
- Create kernel mode stack trace database
- Create user mode stack trace database
- Debug initial command
- Debug WinLogon
- Disable heap coalesce on free
- Disable paging of kernel stacks
- Disable protected DLL verification
- Disable stack extension
- Early critical section event creation
- Enable application verifier
- Enable bad handles detection
- Enable close exception
- Enable debugging of Win32 subsystem
- Enable exception logging
- Enable heap free checking
- Enable heap parameter checking
- Enable heap tagging
- Enable heap tagging by DLL
- Enable heap tail checking
- Enable heap validation on call

Enable loading of kernel debugger symbols
Enable object handle type tagging
Enable page heap
Enable pool tagging
Enable silent process exit monitoring
Enable system critical breaks
Load image using large pages if possible
Maintain a list of objects for each type
Object Reference Tracing
Show loader snaps
Special Pool
Stop on exception
Stop on hung GUI
Stop on unhandled user-mode exception
GFlags Examples
GFlags Examples
Example 1: Displaying Global Flags
Example 2: Setting a Flag by Using a Flag Abbreviation
Example 3: Setting a Flag by Using Its Hexadecimal Value
Example 4: Setting Multiple Flags
Example 5: Clearing a Flag
Example 6: Clearing All Flags
Example 7: Clearing All Flags for an Image File
Example 8: Enlarging the User-Mode Stack Trace Database
Example 9: Detecting a Pool Memory Leak
Example 10: Detecting a Heap Memory Leak in a Process
Example 11: Enabling Page Heap Verification
Example 12: Using Page Heap Verification to Find a Bug
Example 13: Listing Image Files with Global Flags
Example 14: Configuring Special Pool
Example 15: Using Object Reference Tracing
Kill Tool

Logger and LogViewer
Logger and LogViewer
Logger
Logger
Using the Debugger and Logexts.dll
Using Logger.exe
Logger Restrictions and Limitations
LogViewer
LogViewer
Reading the LogViewer Display
Filtering the LogViewer Function List
Exporting LogViewer Files to Text
The Logger Manifest
The Logger Manifest
Overview of the Logging Manifest
Manifest File Placement
Manifest File Format
PLMDebug
Remote Tool
Remote Tool
Remote Tool
Remote Tool Concepts
Remote Tool Commands
Remote Tool Commands
Remote Server Syntax
Remote Client Syntax
Remote Server Query Command
Remote Session Commands
Remote Tool Examples
TList
TList
TList Commands
TList Examples

- UMDH
  - UMDH
    - Preparing to Use UMDH
    - UMDH Commands
      - UMDH Commands
      - Analyze a Running Process
      - Analyze UMDH Logs
    - Interpreting a UMDH Log
    - Interpreting a Log Comparison
  - USBView
- Tools Related to Debugging Tools for Windows
  - Tools Related to Debugging Tools for Windows
  - Windows Error Reporting
- Source Code
  - Source Code
  - Source Path
  - Using a Source Server
  - SrcSrv
    - SrcSrv
    - Using SrcSrv
    - Source Indexing
      - Source Indexing
      - The Srcsrv.ini File
      - The Ssindex.cmd Script
      - The SrcTool Utility
      - The PDBStr Tool
      - The VSSDump Tool
    - Source Control Systems
      - Source Control Systems
      - Using Visual SourceSafe
      - Debugging with Visual SourceSafe
      - Using CVS

Using Other Source Control Systems
Using Other Source Control Systems
Creating Your Own Provider Module
Creating Your Own Source Control System
Language Specification 1
Language Specification 2
HTTP Sites and UNC Shares
HTTP Sites and UNC Shares
Setting Up the Web Site
Extracting Source Files
Modifying the Source Indexing Streams in a .pdb File
Using UNC Shares
Using HTTP Sites and UNC Shares in Conjuction with Regular Version Control
Security Considerations
Security Considerations
Debug Privilege
Security Vulnerabilities
Security Vulnerabilities
Security During Kernel-Mode Debugging
Security During User-Mode Debugging
Security During Postmortem Debugging
Security During Remote Debugging
Secure Mode
Secure Mode
Features of Secure Mode
Activating Secure Mode
Processor Architecture
Processor Architecture
The x86 Processor
The x86 Processor
x86 Architecture
x86 Instructions

- Annotated x86 Disassembly
- The x64 Processor
  - The x64 Processor
  - x64 Architecture
  - x64 Instructions
  - Annotated x64 Disassembly
- Debugger Engine and Extension APIs
  - Debugger Engine and Extension APIs
  - Debugger Engine Introduction
  - Debugger Engine Overview
    - Debugger Engine Overview
    - Debugging Session and Execution Model
    - Client Objects
    - Input and Output
    - Remote Debugging
    - Targets
    - Events
    - Breakpoints
    - Symbols
    - Memory
    - Threads and Processes
  - Using the Debugger Engine API
    - Using the Debugger Engine API
    - Debugger Engine API Overview
      - Debugger Engine API Overview
      - Interacting with the Engine
        - Interacting with the Engine
        - Using Client Objects
        - Using Callback Objects
      - Using Input and Output
      - Monitoring Events
        - Monitoring Events

Event Filters
Event Information
Using Breakpoints
Using Breakpoints
Setting Breakpoints
Controlling Breakpoint Flags and Parameters
Memory Access
Memory Access
Virtual and Physical Memory
Registers
Other Data Spaces
Examining the Stack Trace
Controlling Threads and Processes
Using Symbols
Using Symbols
Modules
Types
Scopes and Symbol Groups
Using Source Files
Connecting to Targets
Connecting to Targets
Live User-Mode Targets
Live Kernel-Mode Targets
Dump-File Targets
Remote Targets
Target Information
Target State
Calling Extensions and Extension Functions
Assembling and Disassembling Instructions
Writing DbgEng Extensions
Writing DbgEng Extensions
DbgEng Extension Design Guide

DbgEng Extension Design Guide
Anatomy of a DbgEng Extension DLL
Using Clients and the Engine
Writing DbgEng Extension Code
Building DbgEng Extensions
EngExtCpp Extensions
EngExtCpp Extensions
EngExtCpp Extension Design Guide
EngExtCpp Extension Design Guide
EngExtCpp Extension Libraries
Client Objects and the Engine
Writing EngExtCpp Extensions
Building EngExtCpp Extensions
Parsing Extension Arguments
Typed Data
Writing WdbgExts Extensions
Writing WdbgExts Extensions
WdbgExts Extension Design Guide
WdbgExts Extension Design Guide
WdbgExts Extension API Overview
32-Bit Pointers and 64-Bit Pointers
Using WdbgExts Extension Callbacks
Using the DECLARE_API Macro
Writing WdbgExts Extension Code
Writing WdbgExts Extension Code
WdbgExts Input and Output
WdbgExts Memory Access
WdbgExts Threads and Processes
WdbgExts Symbols
WdbgExts Target Information
Building WdbgExts Extensions
Writing Custom Analysis Debugger Extensions

- Writing Custom Analysis Debugger Extensions
- Writing an Analysis Extension Plug-in to Extend !analyze
- Metadata Files for Analysis Extension Plug-ins
- Failure Analysis Entries

Debugger Programming Reference

Debug Engine Interfaces

Debugger Engine Reference

- Debugger Engine Reference
- DebugBaseEventCallbacks
- DebugBaseEventCallbacksWide
- Debug API Constants
  - Debug API Constants
  - DBG_ASMOPT_XXX
  - DBG_ATTACH_XXX
  - DBG_DUMP_XXX
  - DBG_DUMP_FIELD_XXX
  - DBG_ENGOPT_XXX
  - DBG_EVENT_XXX
  - DEBUG_FILTER_XXX
  - DEBUG_FORMAT_XXX
  - DEBUG_OUTCB_XXX
  - DEBUG_OUTCTL_XXX
  - DEBUG_OUTPUT_XXX
  - DEBUG_PROCESS_XXX
  - DEBUG_STATUS_XXX
  - DEBUG_SYMBOL_XXX
  - DEBUG_TYPEOPTS_XXX
- HRESULT Values
- Request
  - Request
  - DEBUG_REQUEST_SOURCE_PATH_HAS_SOURCE_SERVER
  - DEBUG_REQUEST_TARGET_EXCEPTION_CONTEXT

DEBUG_REQUEST_TARGET_EXCEPTION_THREAD
DEBUG_REQUEST_TARGET_EXCEPTION_RECORD
DEBUG_REQUEST_GET_ADDITIONAL_CREATE_OPTIONS
DEBUG_REQUEST_SET_ADDITIONAL_CREATE_OPTIONS
DEBUG_REQUEST_GET_WIN32_MAJOR_MINOR_VERSIONS
DEBUG_REQUEST_TARGET_CAN_DETACH
DEBUG_REQUEST_SET_LOCAL_IMPLICIT_COMMAND_LINE
DEBUG_REQUEST_GET_CAPTURED_EVENT_CODE_OFFSET
DEBUG_REQUEST_READ_CAPTURED_EVENT_CODE_STREAM
DEBUG_REQUEST_EXT_TYPED_DATA_ANSI
DEBUG_REQUEST_EXT_TYPED_DATA_ANSI
EXT_TDOP_COPY
EXT_TDOP_RELEASE
EXT_TDOP_SET_FROM_EXPR
EXT_TDOP_SET_FROM_U64_EXPR
EXT_TDOP_GET_FIELD
EXT_TDOP_EVALUATE
EXT_TDOP_GET_TYPE_NAME
EXT_TDOP_OUTPUT_TYPE_NAME
EXT_TDOP_OUTPUT_SIMPLE_VALUE
EXT_TDOP_OUTPUT_FULL_VALUE
EXT_TDOP_HAS_FIELD
EXT_TDOP_GET_FIELD_OFFSET
EXT_TDOP_GET_ARRAY_ELEMENT
EXT_TDOP_GET_DEREFERENCE
EXT_TDOP_GET_TYPE_SIZE
EXT_TDOP_OUTPUT_TYPE_DEFINITION
EXT_TDOP_GET_POINTER_TO
EXT_TDOP_SET_FROM_TYPE_ID_AND_U64
EXT_TDOP_SET_PTR_FROM_TYPE_ID_AND_U64
ExtExtension
Specific Exceptions

WdbgExts Functions
Customizing Debugger Output Using DML
JavaScript Debugger Scripting
  JavaScript Debugger Scripting
  JavaScript Debugger Example Scripts
  Native Debugger Objects in JavaScript Extensions
  Native Debugger Objects in JavaScript Extensions - Debugger Object Details
  Native Debugger Objects in JavaScript Extensions - Design Considerations
Native Debugger Objects in NatVis
Using LINQ With the debugger objects
Debugger Data Model Function Aliases
Debugger Data Model C++ Overview
  Debugger Data Model C++ Overview
  Debugger Data Model C++ Interfaces
  Debugger Data Model C++ Objects
  Debugger Data Model C++ Additional Interfaces
  Debugger Data Model C++ Concepts
  Debugger Data Model C++ Scripting
Debugger Data Model - Code Namespace
  Debugger Data Model - Code Namespace
  Debugger Data Model - Disassembler Objects
  Debugger Data Model - Basic Block Objects
  Debugger Data Model - Instruction Objects
  Debugger Data Model - Instruction Attributes Objects
  Debugger Data Model - Operand Objects
  Debugger Data Model - Operand Attributes Objects
  Debugger Data Model - Register Objects
  Debugger Data Model - Live Variable Objects
  Debugger Data Model - Source Information Objects
  Debugger Data Model - Control Flow Objects
Debugger Data Model - Collections Namespace
Debugger Data Model - File System Namespace

Debugger Data Model - File System Namespace
Debugger Data Model - Directory Objects
Debugger Data Model - File Objects
Debugger Data Model - Text Reader Objects
Debugger Data Model - Text Writer Objects
Glossary
Glossary
A
B
C
D
E
F
H
I
K
L
M
N
O
P
R
S
T
U
V
W
Debugger Operation
Debugger Operation
Debugging Using WinDbg
Debugging Using WinDbg
Debugging a User-Mode Process Using WinDbg

- Debugging a UWP app using WinDbg
- Opening a Dump File Using WinDbg
- Live Kernel-Mode Debugging Using WinDbg
- Ending a Debugging Session in WinDbg
- Setting Symbol and Executable Image Paths in WinDbg
- Remote Debugging Using WinDbg
- Entering Debugger Commands in WinDbg
- Using the Command Browser Window in WinDbg
- Setting Breakpoints in WinDbg
- Viewing the Call Stack in WinDbg
- Assembly Code Debugging in WinDbg
- Source Code Debugging in WinDbg
- Viewing and Editing Memory in WinDbg
- Viewing and Editing Global Variables in WinDbg
- Viewing and Editing Local Variables in WinDbg
- Viewing and Editing Registers in WinDbg
- Controlling Processes and Threads in WinDbg
- Configuring Exceptions and Events in WinDbg
- Keeping a Log File in WinDbg
- Using the Watch Window
- Using the Scratch Pad

Debugging Using KD and NTKD

- Debugging Using KD and NTKD
- Opening a Dump File Using KD
- Live Kernel-Mode Debugging Using KD
- Ending a Debugging Session in KD
- Setting Symbol and Executable Image Paths in KD
- Setting Breakpoints in KD
- Viewing the Call Stack in KD
- Viewing and Editing Memory in KD
- Viewing and Editing Registers in KD
- Remote Debugging Using KD

- Configuring Exceptions and Events in KD
- Keeping a Log File in KD

Debugging Using CDB and NTSD

- Debugging Using CDB and NTSD
- Debugging a User-Mode Process Using CDB
- Opening a Dump File Using CDB
- Ending a Debugging Session in CDB
- Setting Symbol and Executable Image Paths in CDB
- Setting Breakpoints in CDB
- Viewing the Call Stack in CDB
- Viewing and Editing Memory in CDB
- Viewing and Editing Registers in CDB
- Configuring Exceptions and Events in CDB
- Keeping a Log File in CDB

Local Kernel-Mode Debugging

Controlling the Target

Enabling Postmortem Debugging

Using the Debugger Command Window

- Using the Debugger Command Window
- Using Debugger Commands
- Evaluating Expressions
- Using Shell Commands
- Using Aliases
- Using Script Files
- Using Debugger Command Programs
  - Using Debugger Command Programs
  - Elements of a Debugger Command Program
  - Control Flow Tokens
  - Executing a Debugger Command Program
  - Debugger Command Program Examples

Using the WinDbg Graphical Interface

- Using the WinDbg Graphical Interface

- Using Debugging Information Windows
  - Using Debugging Information Windows
  - Opening a Window
  - Closing a Window
  - Configuring a Window
  - Moving Through a Window
  - Cutting and Pasting Text
  - Changing Text Properties
  - Positioning the Windows
    - Positioning the Windows
    - Debugging with Floating and Docked Windows
    - Docking a Window
    - Tabbing a Window
    - Undocking a Window
    - Creating a New Dock
    - Resizing and Moving a Window
    - Arranging Windows
- Using Workspaces
  - Using Workspaces
  - Creating and Opening a Workspace
  - Workspace Contents
  - Using and Customizing WinDbg Themes
    - Using and Customizing WinDbg Themes
    - Loading a Theme
    - Customizing a Theme
    - Using Themes Provided in Debugging Tools for Windows
- Using the Toolbar and Status Bar
- Using the Help Documentation

Using Debugger Extensions

- Using Debugger Extensions
- Loading Debugger Extension DLLs
- Using Debugger Extension Commands

- Writing New Debugger Extensions
- Remote Debugging
  - Remote Debugging
  - Choosing the Best Remote Debugging Method
  - Remote Debugging Through the Debugger
    - Remote Debugging Through the Debugger
    - Activating a Debugging Server
    - Searching for Debugging Servers
    - Activating a Debugging Client
    - Client and Server Examples
    - Controlling a Remote Debugging Session
  - Controlling the User-Mode Debugger from the Kernel Debugger
    - Controlling the User-Mode Debugger from the Kernel Debugger
    - Starting the Debugging Session
    - Switching Modes
    - When to Use This Technique
    - Combining This Method with Remote Debugging
  - Remote Debugging Through Remote.exe
    - Remote Debugging Through Remote.exe
    - The Remote.exe Utility
    - Starting a Remote.exe Session
    - Remote.exe Batch Files
  - Process Servers (User Mode)
    - Process Servers (User Mode)
    - Activating a Process Server
    - Searching for Process Servers
    - Activating a Smart Client
    - Process Server Examples
    - Controlling a Process Server Session
  - KD Connection Servers (Kernel Mode)
    - KD Connection Servers (Kernel Mode)
    - Activating a KD Connection Server

Searching for KD Connection Servers
Activating a Smart Client (Kernel Mode)
KD Connection Server Examples
Controlling a KD Connection Server Session
Repeaters
Repeaters
Activating a Repeater
Using a Repeater
Repeater Examples
Advanced Remote Debugging Scenarios
Advanced Remote Debugging Scenarios
Debugging Targets on Multiple Computers
Symbols in the Middle
Two Firewalls
Remote Debugging on Workgroup Computers
Debugging Previous Versions of Windows
Debugging Previous Versions of Windows
Debugging Tools For Windows: What's New
Debugging Tools For Windows: What's New
Debugging Tools for Windows: New for Windows 8.1
Debugging Tools for Windows: New for Windows 8
Debugging Tools For Windows8 Release Notes
Debugging Windows Vista
Debugging Using Visual Studio
Setting Up User-Mode Debugging in Visual Studio
Setting Up User-Mode Debugging in Visual Studio
Debugging a User-Mode Process Using Visual Studio
Opening a Dump File Using Visual Studio
Kernel-Mode Debugging in Visual Studio
Ending a Debugging Session in Visual Studio
Setting Symbol and Executable Image Paths in Visual Studio
Remote Debugging Using Visual Studio

Entering Debugger Commands in Visual Studio
Setting Breakpoints in Visual Studio
Viewing the Call Stack in Visual Studio
Source Code Debugging in Visual Studio
Viewing and Editing Memory and Registers in Visual Studio
Controlling Threads and Processes in Visual Studio
Configuring Exceptions and Events in Visual Studio
Keeping a Log File in Visual Studio
Setting Up Kernel-Mode Debugging in Visual Studio
Setting Up Kernel-Mode Debugging in Visual Studio
Setting Up Kernel-Mode Debugging over a Network Cable in Visual Studio
Setting Up Kernel-Mode Debugging over a 1394 Cable in Visual Studio
Setting Up Kernel-Mode Debugging over a USB 3.0 Cable in Visual Studio
Setting Up Kernel-Mode Debugging over a USB 2.0 Cable in Visual Studio
Setting Up Kernel-Mode Debugging over a Serial Cable in Visual Studio
Setting Up Kernel-Mode Debugging using Serial over USB in Visual Studio
Setting Up Kernel-Mode Debugging of a Virtual Machine in Visual Studio
Debugging Techniques
Debugging Techniques
Standard Debugging Techniques
Standard Debugging Techniques
Using Breakpoints
Using Breakpoints
Methods of Controlling Breakpoints
Breakpoint Syntax
Unresolved Breakpoints (bu Breakpoints)
Processor Breakpoints (ba Breakpoints)
Initial Breakpoint
User Space and System Space
Risks Entailed When Setting Breakpoints
Conditional breakpoints in WinDbg and other Windows debuggers
Executing Until a Specified State is Reached

Reading and Writing Memory
  Reading and Writing Memory
  Accessing Memory by Virtual Address
  Accessing Memory by Physical Address
  Accessing Global Variables
  Accessing Local Variables
  Controlling Variables Through the Watch Window
  Converting Virtual Addresses to Physical Addresses
Using the !analyze Extension
Handling a Bug Check When Driver Verifier is Enabled
Noninvasive Debugging (User Mode)
Debugging in Assembly Mode
Debugging in Source Mode
Debugging Optimized Code and Inline Functions
Debugging Managed Code Using the Windows Debugger
Debugging UWP Apps Using the Windows Debugger
Changing Contexts
Controlling Processes and Threads
Using Debugger Markup Language
Controlling Exceptions and Events
Finding the Process ID
Debugging a Stack Overflow
Manually Walking a Stack
Debugging a Stack Trace that has JScript Frames
Debugging an Application Failure
Reattaching to the Target Application
Crashing and Rebooting the Target Computer
Synchronizing with the Target Computer
Finding a Memory Leak
  Finding a Memory Leak
  Determining Whether a Leak Exists
  Finding a Kernel-Mode Memory Leak

Finding a Kernel-Mode Memory Leak
Using PoolMon to Find a Kernel-Mode Memory Leak
Using the Kernel Debugger to Find a Kernel-Mode Memory Leak
Using Driver Verifier to Find a Kernel-Mode Memory Leak
Finding a User-Mode Memory Leak
Finding a User-Mode Memory Leak
Using Performance Monitor to Find a User-Mode Memory Leak
Using UMDH to Find a User-Mode Memory Leak
Debugging a Time Out
Debugging a Time Out
Resource Time Outs
Critical Section Time Outs
Debugging a Stalled System
Debugging a Stalled System
Finding the Failed Process
Debugging an Interrupt Storm
Debugging Multiple Targets
Tracking Down a Processor Hog
Determining the ACL of an Object
Displaying a Critical Section
Debugging a Deadlock
Debugging a Failed Driver Unload
Reading Bug Check Callback Data
Debugging a User-Mode Failure with KD
Crashing and Rebooting the Target Computer
Mapping Driver Files
Messages from the Target
Messages from the Target
Breaking Into the Debugger
Sending Output to the Debugger
Reading and Filtering Debugging Messages
Determining if a Debugger is Attached

Specialized Debugging Techniques
Specialized Debugging Techniques
Debugging ARM64
Windows Runtime Debugging
Kernel-Mode Driver Framework Debugging
User-Mode Driver Framework Debugging
Debugging Device Nodes and Device Stacks
Debugging Plug and Play and Power Issues
Debugging a User-Mode Failure with KD
Debugging a Device Installation Co-Installer
Debugging a Dual-Boot Machine
Debugging Windows Setup and the OS Loader
Debugging CSRSS
Debugging WinLogon
Debugging BIOS Code
Specifying Module and Function Owners
RPC Debugging
RPC Debugging
Overview of RPC Debugging
Enabling RPC State Information
Displaying RPC State Information
Displaying RPC State Information
Using the RPC Debugger Extensions
Using the DbgRpc Tool
Get RPC Cell Information
Get RPC Endpoint Information
Get RPC Thread Information
Get RPC Call Information
Get RPC Client Call Information
Common RPC Debugging Techniques
Common RPC Debugging Techniques
Analyzing a Stuck Call Problem

Tracking Contention in the Server Process
Checking for Stuck Threads
Identifying the Caller From the Server Thread
RPC State Information Internals
ACPI Debugging
ACPI Debugging
The AMLI Debugger
The AMLI Debugger
Introduction to the AMLI Debugger
Setting Up an AML Debugging Session
Basic AML Debugging
Using AMLI Debugger Extensions
Using AMLI Debugger Commands
AML Debugging Examples
Other ACPI Debugging Extensions
NDIS Debugging
NDIS Debugging
Overview of NDIS Debugging
Preparing for NDIS Debugging
Enabling NDIS Debug Tracing
Enabling NDIS Debug Tracing
Enabling NDIS Debug Tracing By Setting Registry Values
Kernel Streaming Debugging
Kernel Streaming Debugging
Overview of Kernel Streaming Debugging
Analyzing a Video Stream Stall
Analyzing a Video Stream Stall
Determining the Cause of a Video Stream Stall
Debugging a Processing Stall
Using Logging to Track Important Events
Interpreting Bug Check 0xCB
Analyzing a Capture Stall

Live Local Debugging
Graph Analysis with Unloadable Modules
Using !ks.graph
SCSI Miniport Debugging
SCSI Miniport Debugging
Overview of SCSI Miniport Debugging
Extensions for Debugging SCSI Miniport Drivers
Bug Checks for SCSI Miniport Debugging
Analyzing Stalled Drivers and Time-Outs
Important Breakpoints for Analyzing Reproducible Problems
Plug and Play Debugging
Plug and Play Debugging
Extensions for Debugging Plug and Play Drivers
Determining the Status of a Device
Device Node Status Flags
Device Manager Problem Codes
Checking for Resource Conflicts
Debugging a Service Application
Debugging a Service Application
Choosing the Best Method
Preparing to Debug the Service Application
Debugging the Service Application Automatically
Debugging the Service Application Manually
Symbols for Windows Debugging (WinDbg, KD, CDB, NTSD)
Symbols for Windows Debugging (WinDbg, KD, CDB, NTSD)
Introduction to Symbols
Introduction to Symbols
Symbol path for Windows debuggers
Symbols and Symbol Files
Public and Private Symbols
Accessing Symbols for Debugging
Accessing Symbols for Debugging

- Installing Windows Symbol Files
- Symbol Stores and Symbol Servers
  - Symbol Stores and Symbol Servers
  - Using a Symbol Server
  - SymSrv
    - SymSrv
    - Microsoft public symbol server
    - Advanced SymSrv Use
    - Firewalls and Proxy Servers
  - HTTP Symbol Stores
  - File Share (SMB) Symbol Server
  - Symbol Store Folder Tree
  - SymProxy
    - SymProxy
    - Installing SymProxy
    - Configuring the Registry
    - Choosing Network Security Credentials
    - Configuring IIS for SymProxy
    - Setting Up Exclusion Lists
    - Dealing with Unavailable Symbol Stores
    - Handling File Pointers
    - Caching Acquired Symbol Files
    - SymProxy Automated Installation
  - Other Symbol Stores
  - Other Symbol Servers
- Deferred Symbol Loading
- Avoiding debugger searches for unneeded symbols

SymStore
- SymStore
- SymStore Transactions
- File System References and Symbol Files
- SymStore Compressed Files

Symbol Storage Format
How the Debugger Recognizes Symbols
How the Debugger Recognizes Symbols
Symbol Syntax and Symbol Matching
Symbol Options
Symbol Status Abbreviations
Symbol Problems While Debugging
Symbol Problems While Debugging
Verifying Symbols
Matching Symbol Names
Reading Symbols from Paged-Out Headers
Mapping Symbols When the PEB is Paged Out
Debugging User-Mode Processes Without Symbols
Debugging Performance-Optimized Code
Offline Symbols for Windows Update
AgeStore
AgeStore
Using AgeStore
AgeStore Command-Line Options
DBH
DBH
Using DBH
Additional DBH Examples
DBH Command-Line Options
DBH Commands
PDBCopy
PDBCopy
Using PDBCopy
Choosing Which Public Symbols to Remove
PDBCopy Command-Line Options
SymChk
SymChk

- Using SymChk
- Using a Manifest File with SymChk
- SymChk Command-Line Options

Crash dump analysis using the Windows debuggers (WinDbg)

- Crash dump analysis using the Windows debuggers (WinDbg)
- Kernel-Mode Dump Files
  - Kernel-Mode Dump Files
  - Varieties of Kernel-Mode Dump Files
    - Varieties of Kernel-Mode Dump Files
    - Complete Memory Dump
    - Kernel Memory Dump
    - Small Memory Dump
    - Automatic Memory Dump
    - Active Memory Dump
  - Creating a Kernel-Mode Dump File
    - Creating a Kernel-Mode Dump File
    - Enabling a Kernel-Mode Dump File
    - Forcing a System Crash
      - Forcing a System Crash
      - Forcing a System Crash from the Debugger
      - Forcing a System Crash from the Keyboard
    - Creating a Dump File Without a System Crash
    - Verifying the Creation of a Kernel-Mode Dump File
  - Analyzing a Kernel-Mode Dump File
    - Analyzing a Kernel-Mode Dump File
    - Analyzing a Kernel-Mode Dump File with KD
    - Analyzing a Kernel-Mode Dump File with WinDbg
    - Analyzing a Kernel-Mode Dump File with KAnalyze
- User-Mode Dump Files
  - User-Mode Dump Files
  - Analyzing a User-Mode Dump File
- Extracting Information from a Dump File

- CAB Files that Contain Paging Files Along with a Memory Dump
- Debugging OCA minidump files

Bug Checks (Blue Screens)

- Bug Checks (Blue Screens)
- General Tips for Blue Screens
- Blue Screen Data
- Bug Check Code Reference
  - Bug Check Code Reference
  - Bug Check 0x1: APC_INDEX_MISMATCH
  - Bug Check 0x2: DEVICE_QUEUE_NOT_BUSY
  - Bug Check 0x3: INVALID_AFFINITY_SET
  - Bug Check 0x4: INVALID_DATA_ACCESS_TRAP
  - Bug Check 0x5: INVALID_PROCESS_ATTACH_ATTEMPT
  - Bug Check 0x6: INVALID_PROCESS_DETACH_ATTEMPT
  - Bug Check 0x7: INVALID_SOFTWARE_INTERRUPT
  - Bug Check 0x8: IRQL_NOT_DISPATCH_LEVEL
  - Bug Check 0x9: IRQL_NOT_GREATER_OR_EQUAL
  - Bug Check 0xA: IRQL_NOT_LESS_OR_EQUAL
  - Bug Check 0xB: NO_EXCEPTION_HANDLING_SUPPORT
  - Bug Check 0xC: MAXIMUM_WAIT_OBJECTS_EXCEEDED
  - Bug Check 0xD: MUTEX_LEVEL_NUMBER_VIOLATION
  - Bug Check 0xE: NO_USER_MODE_CONTEXT
  - Bug Check 0xF: SPIN_LOCK_ALREADY_OWNED
  - Bug Check 0x10: SPIN_LOCK_NOT_OWNED
  - Bug Check 0x11: THREAD_NOT_MUTEX_OWNER
  - Bug Check 0x12: TRAP_CAUSE_UNKNOWN
  - Bug Check 0x13: EMPTY_THREAD_REAPER_LIST
  - Bug Check 0x14: CREATE_DELETE_LOCK_NOT_LOCKED
  - Bug Check 0x15: LAST_CHANCE_CALLED_FROM_KMODE
  - Bug Check 0x16: CID_HANDLE_CREATION
  - Bug Check 0x17: CID_HANDLE_DELETION
  - Bug Check 0x18: REFERENCE_BY_POINTER

Bug Check 0x19: BAD_POOL_HEADER

Bug Check 0x1A: MEMORY_MANAGEMENT

Bug Check 0x1B: PFN_SHARE_COUNT

Bug Check 0x1C: PFN_REFERENCE_COUNT

Bug Check 0x1D: NO_SPIN_LOCK_AVAILABLE

Bug Check 0x1E: KMODE_EXCEPTION_NOT_HANDLED

Bug Check 0x1F: SHARED_RESOURCE_CONV_ERROR

Bug Check 0x20: KERNEL_APC_PENDING_DURING_EXIT

Bug Check 0x21: QUOTA_UNDERFLOW

Bug Check 0x22: FILE_SYSTEM

Bug Check 0x23: FAT_FILE_SYSTEM

Bug Check 0x24: NTFS_FILE_SYSTEM

Bug Check 0x25: NPFS_FILE_SYSTEM

Bug Check 0x26: CDFS_FILE_SYSTEM

Bug Check 0x27: RDR_FILE_SYSTEM

Bug Check 0x28: CORRUPT_ACCESS_TOKEN

Bug Check 0x29: SECURITY_SYSTEM

Bug Check 0x2A: INCONSISTENT_IRP

Bug Check 0x2B: PANIC_STACK_SWITCH

Bug Check 0x2C: PORT_DRIVER_INTERNAL

Bug Check 0x2D: SCSI_DISK_DRIVER_INTERNAL

Bug Check 0x2E: DATA_BUS_ERROR

Bug Check 0x2F: INSTRUCTION_BUS_ERROR

Bug Check 0x30: SET_OF_INVALID_CONTEXT

Bug Check 0x31: PHASE0_INITIALIZATION_FAILED

Bug Check 0x32: PHASE1_INITIALIZATION_FAILED

Bug Check 0x33: UNEXPECTED_INITIALIZATION_CALL

Bug Check 0x34: CACHE_MANAGER

Bug Check 0x35: NO_MORE_IRP_STACK_LOCATIONS

Bug Check 0x36: DEVICE_REFERENCE_COUNT_NOT_ZERO

Bug Check 0x37: FLOPPY_INTERNAL_ERROR

Bug Check 0x38: SERIAL_DRIVER_INTERNAL

Bug Check 0x39: SYSTEM_EXIT_OWNED_MUTEX

Bug Check 0x3A: SYSTEM_UNWIND_PREVIOUS_USER

Bug Check 0x3B: SYSTEM_SERVICE_EXCEPTION

Bug Check 0x3C: INTERRUPT_UNWIND_ATTEMPTED

Bug Check 0x3D: INTERRUPT_EXCEPTION_NOT_HANDLED

Bug Check 0x3E: MULTIPROCESSOR_CONFIGURATION_NOT_SUPPORTED

Bug Check 0x3F: NO_MORE_SYSTEM_PTES

Bug Check 0x40: TARGET_MDL_TOO_SMALL

Bug Check 0x41: MUST_SUCCEED_POOL_EMPTY

Bug Check 0x42: ATDISK_DRIVER_INTERNAL

Bug Check 0x43: NO_SUCH_PARTITION

Bug Check 0x44: MULTIPLE_IRP_COMPLETE_REQUESTS

Bug Check 0x45: INSUFFICIENT_SYSTEM_MAP_REGS

Bug Check 0x46: DEREF_UNKNOWN_LOGON_SESSION

Bug Check 0x47: REF_UNKNOWN_LOGON_SESSION

Bug Check 0x48: CANCEL_STATE_IN_COMPLETED_IRP

Bug Check 0x49: PAGE_FAULT_WITH_INTERRUPTS_OFF

Bug Check 0x4A: IRQL_GT_ZERO_AT_SYSTEM_SERVICE

Bug Check 0x4B: STREAMS_INTERNAL_ERROR

Bug Check 0x4C: FATAL_UNHANDLED_HARD_ERROR

Bug Check 0x4D: NO_PAGES_AVAILABLE

Bug Check 0x4E: PFN_LIST_CORRUPT

Bug Check 0x4F: NDIS_INTERNAL_ERROR

Bug Check 0x50: PAGE_FAULT_IN_NONPAGED_AREA

Bug Check 0x51: REGISTRY_ERROR

Bug Check 0x52: MAILSLOT_FILE_SYSTEM

Bug Check 0x53: NO_BOOT_DEVICE

Bug Check 0x54: LM_SERVER_INTERNAL_ERROR

Bug Check 0x55: DATA_COHERENCY_EXCEPTION

Bug Check 0x56: INSTRUCTION_COHERENCY_EXCEPTION

Bug Check 0x57: XNS_INTERNAL_ERROR

Bug Check 0x58: FTDISK_INTERNAL_ERROR

Bug Check 0x59: PINBALL_FILE_SYSTEM

Bug Check 0x5A: CRITICAL_SERVICE_FAILED

Bug Check 0x5B: SET_ENV_VAR_FAILED

Bug Check 0x5C: HAL_INITIALIZATION_FAILED

Bug Check 0x5D: UNSUPPORTED_PROCESSOR

Bug Check 0x5E: OBJECT_INITIALIZATION_FAILED

Bug Check 0x5F: SECURITY_INITIALIZATION_FAILED

Bug Check 0x60: PROCESS_INITIALIZATION_FAILED

Bug Check 0x61: HAL1_INITIALIZATION_FAILED

Bug Check 0x62: OBJECT1_INITIALIZATION_FAILED

Bug Check 0x63: SECURITY1_INITIALIZATION_FAILED

Bug Check 0x64: SYMBOLIC_INITIALIZATION_FAILED

Bug Check 0x65: MEMORY1_INITIALIZATION_FAILED

Bug Check 0x66: CACHE_INITIALIZATION_FAILED

Bug Check 0x67: CONFIG_INITIALIZATION_FAILED

Bug Check 0x68: FILE_INITIALIZATION_FAILED

Bug Check 0x69: IO1_INITIALIZATION_FAILED

Bug Check 0x6A: LPC_INITIALIZATION_FAILED

Bug Check 0x6B: PROCESS1_INITIALIZATION_FAILED

Bug Check 0x6C: REFMON_INITIALIZATION_FAILED

Bug Check 0x6D: SESSION1_INITIALIZATION_FAILED

Bug Check 0x6E: SESSION2_INITIALIZATION_FAILED

Bug Check 0x6F: SESSION3_INITIALIZATION_FAILED

Bug Check 0x70: SESSION4_INITIALIZATION_FAILED

Bug Check 0x71: SESSION5_INITIALIZATION_FAILED

Bug Check 0x72: ASSIGN_DRIVE_LETTERS_FAILED

Bug Check 0x73: CONFIG_LIST_FAILED

Bug Check 0x74: BAD_SYSTEM_CONFIG_INFO

Bug Check 0x75: CANNOT_WRITE_CONFIGURATION

Bug Check 0x76: PROCESS_HAS_LOCKED_PAGES

Bug Check 0x77: KERNEL_STACK_INPAGE_ERROR

Bug Check 0x78: PHASE0_EXCEPTION

Bug Check 0x79: MISMATCHED_HAL

Bug Check 0x7A: KERNEL_DATA_INPAGE_ERROR

Bug Check 0x7B: INACCESSIBLE_BOOT_DEVICE

Bug Check 0x7C: BUGCODE_NDIS_DRIVER

Bug Check 0x7D: INSTALL_MORE_MEMORY

Bug Check 0x7E: SYSTEM_THREAD_EXCEPTION_NOT_HANDLED

Bug Check 0x7F: UNEXPECTED_KERNEL_MODE_TRAP

Bug Check 0x80: NMI_HARDWARE_FAILURE

Bug Check 0x81: SPIN_LOCK_INIT_FAILURE

Bug Check 0x82: DFS_FILE_SYSTEM

Bug Check 0x85: SETUP_FAILURE

Bug Check 0x8B: MBR_CHECKSUM_MISMATCH

Bug Check 0x8E: KERNEL_MODE_EXCEPTION_NOT_HANDLED

Bug Check 0x8F: PP0_INITIALIZATION_FAILED

Bug Check 0x90: PP1_INITIALIZATION_FAILED

Bug Check 0x92: UP_DRIVER_ON_MP_SYSTEM

Bug Check 0x93: INVALID_KERNEL_HANDLE

Bug Check 0x94: KERNEL_STACK_LOCKED_AT_EXIT

Bug Check 0x96: INVALID_WORK_QUEUE_ITEM

Bug Check 0x97: BOUND_IMAGE_UNSUPPORTED

Bug Check 0x98: END_OF_NT_EVALUATION_PERIOD

Bug Check 0x99: INVALID_REGION_OR_SEGMENT

Bug Check 0x9A: SYSTEM_LICENSE_VIOLATION

Bug Check 0x9B: UDFS_FILE_SYSTEM

Bug Check 0x9C: MACHINE_CHECK_EXCEPTION

Bug Check 0x9E: USER_MODE_HEALTH_MONITOR

Bug Check 0x9F: DRIVER_POWER_STATE_FAILURE

Bug Check 0xA0: INTERNAL_POWER_ERROR

Bug Check 0xA1: PCI_BUS_DRIVER_INTERNAL

Bug Check 0xA2: MEMORY_IMAGE_CORRUPT

Bug Check 0xA3: ACPI_DRIVER_INTERNAL

Bug Check 0xA4: CNSS_FILE_SYSTEM_FILTER

Bug Check 0xA5: ACPI_BIOS_ERROR

Bug Check 0xA7: BAD_EXHANDLE

Bug Check 0xAB: SESSION_HAS_VALID_POOL_ON_EXIT

Bug Check 0xAC: HAL_MEMORY_ALLOCATION

Bug Check 0xAD: VIDEO_DRIVER_DEBUG_REPORT_REQUEST

Bug Check 0xB1: BGI_DETECTED_VIOLATION

Bug Check 0xB4: VIDEO_DRIVER_INIT_FAILURE

Bug Check 0xB8: ATTEMPTED_SWITCH_FROM_DPC

Bug Check 0xB9: CHIPSET_DETECTED_ERROR

Bug Check 0xBA: SESSION_HAS_VALID_VIEWS_ON_EXIT

Bug Check 0xBB: NETWORK_BOOT_INITIALIZATION_FAILED

Bug Check 0xBC: NETWORK_BOOT_DUPLICATE_ADDRESS

Bug Check 0xBD: INVALID_HIBERNATED_STATE

Bug Check 0xBE: ATTEMPTED_WRITE_TO_READONLY_MEMORY

Bug Check 0xBF: MUTEX_ALREADY_OWNED

Bug Check 0xC1: SPECIAL_POOL_DETECTED_MEMORY_CORRUPTION

Bug Check 0xC2: BAD_POOL_CALLER

Bug Check 0xC4: DRIVER_VERIFIER_DETECTED_VIOLATION

Bug Check 0xC5: DRIVER_CORRUPTED_EXPOOL

Bug Check 0xC6: DRIVER_CAUGHT_MODIFYING_FREED_POOL

Bug Check 0xC7: TIMER_OR_DPC_INVALID

Bug Check 0xC8: IRQL_UNEXPECTED_VALUE

Bug Check 0xC9: DRIVER_VERIFIER_IOMANAGER_VIOLATION

Bug Check 0xCA: PNP_DETECTED_FATAL_ERROR

Bug Check 0xCB: DRIVER_LEFT_LOCKED_PAGES_IN_PROCESS

Bug Check 0xCC: PAGE_FAULT_IN_FREED_SPECIAL_POOL

Bug Check 0xCD: PAGE_FAULT_BEYOND_END_OF_ALLOCATION

Bug Check 0xCE: DRIVER_UNLOADED_WITHOUT_CANCELLING_PENDING_OPERATIONS

Bug Check 0xCF: TERMINAL_SERVER_DRIVER_MADE_INCORRECT_MEMORY_REFERENCE

Bug Check 0xD0: DRIVER_CORRUPTED_MMPOOL

Bug Check 0xD1: DRIVER_IRQL_NOT_LESS_OR_EQUAL

Bug Check 0xD2: BUGCODE_ID_DRIVER

Bug Check 0xD3: DRIVER_PORTION_MUST_BE_NONPAGED

Bug Check 0xD4: SYSTEM_SCAN_AT_RAISED_IRQL_CAUGHT_IMPROPER_DRIVER_UNLOAD

Bug Check 0xD5: DRIVER_PAGE_FAULT_IN_FREED_SPECIAL_POOL

Bug Check 0xD6: DRIVER_PAGE_FAULT_BEYOND_END_OF_ALLOCATION

Bug Check 0xD7: DRIVER_UNMAPPING_INVALID_VIEW

Bug Check 0xD8: DRIVER_USED_EXCESSIVE_PTES

Bug Check 0xD9: LOCKED_PAGES_TRACKER_CORRUPTION

Bug Check 0xDA: SYSTEM_PTE_MISUSE

Bug Check 0xDB: DRIVER_CORRUPTED_SYSPTES

Bug Check 0xDC: DRIVER_INVALID_STACK_ACCESS

Bug Check 0xDE: POOL_CORRUPTION_IN_FILE_AREA

Bug Check 0xDF: IMPERSONATING_WORKER_THREAD

Bug Check 0xE0: ACPI_BIOS_FATAL_ERROR

Bug Check 0xE1: WORKER_THREAD_RETURNED_AT_BAD_IRQL

Bug Check 0xE2: MANUALLY_INITIATED_CRASH

Bug Check 0xE3: RESOURCE_NOT_OWNED

Bug Check 0xE4: WORKER_INVALID

Bug Check 0xE6: DRIVER_VERIFIER_DMA_VIOLATION

Bug Check 0xE7: INVALID_FLOATING_POINT_STATE

Bug Check 0xE8: INVALID_CANCEL_OF_FILE_OPEN

Bug Check 0xE9: ACTIVE_EX_WORKER_THREAD_TERMINATION

Bug Check 0xEA: THREAD_STUCK_IN_DEVICE_DRIVER

Bug Check 0xEB: DIRTY_MAPPED_PAGES_CONGESTION

Bug Check 0xEC: SESSION_HAS_VALID_SPECIAL_POOL_ON_EXIT

Bug Check 0xED: UNMOUNTABLE_BOOT_VOLUME

Bug Check 0xEF: CRITICAL_PROCESS_DIED

Bug Check 0xF0: STORAGE_MINIPORT_ERROR

Bug Check 0xF1: SCSI_VERIFIER_DETECTED_VIOLATION

Bug Check 0xF2: HARDWARE_INTERRUPT_STORM

Bug Check 0xF3: DISORDERLY_SHUTDOWN

Bug Check 0xF4: CRITICAL_OBJECT_TERMINATION

Bug Check 0xF5: FLTMGR_FILE_SYSTEM

Bug Check 0xF6: PCI_VERIFIER_DETECTED_VIOLATION

Bug Check 0xF7: DRIVER_OVERRAN_STACK_BUFFER

Bug Check 0xF8: RAMDISK_BOOT_INITIALIZATION_FAILED

Bug Check 0xF9: DRIVER_RETURNED_STATUS_REPARSE_FOR_VOLUME_OPEN

Bug Check 0xFA: HTTP_DRIVER_CORRUPTED

Bug Check 0xFC: ATTEMPTED_EXECUTE_OF_NOEXECUTE_MEMORY

Bug Check 0xFD: DIRTY_NOWRITE_PAGES_CONGESTION

Bug Check 0xFE: BUGCODE_USB_DRIVER

Bug Check 0xFF: RESERVE_QUEUE_OVERFLOW

Bug Check 0x100: LOADER_BLOCK_MISMATCH

Bug Check 0x101: CLOCK_WATCHDOG_TIMEOUT

Bug Check 0x102: DPC_WATCHDOG_TIMEOUT

Bug Check 0x103: MUP_FILE_SYSTEM

Bug Check 0x104: AGP_INVALID_ACCESS

Bug Check 0x105: AGP_GART_CORRUPTION

Bug Check 0x106: AGP_ILLEGALLY_REPROGRAMMED

Bug Check 0x108: THIRD_PARTY_FILE_SYSTEM_FAILURE

Bug Check 0x109: CRITICAL_STRUCTURE_CORRUPTION

Bug Check 0x10A: APP_TAGGING_INITIALIZATION_FAILED

Bug Check 0x10C: FSRTL_EXTRA_CREATE_PARAMETER_VIOLATION

Bug Check 0x10D: WDF_VIOLATION

Bug Check 0x10E: VIDEO_MEMORY_MANAGEMENT_INTERNAL

Bug Check 0x10F: RESOURCE_MANAGER_EXCEPTION_NOT_HANDLED

Bug Check 0x111: RECURSIVE_NMI

Bug Check 0x112: MSRPC_STATE_VIOLATION

Bug Check 0x113: VIDEO_DXGKRNL_FATAL_ERROR

Bug Check 0x114: VIDEO_SHADOW_DRIVER_FATAL_ERROR

Bug Check 0x115: AGP_INTERNAL

Bug Check 0x116: VIDEO_TDR_FAILURE

Bug Check 0x117: VIDEO_TDR_TIMEOUT_DETECTED

Bug Check 0x119: VIDEO_SCHEDULER_INTERNAL_ERROR

Bug Check 0x11A: EM_INITIALIZATION_FAILURE

Bug Check 0x11B: DRIVER_RETURNED_HOLDING_CANCEL_LOCK

Bug Check 0x11C: ATTEMPTED_WRITE_TO_CM_PROTECTED_STORAGE

Bug Check 0x11D: EVENT_TRACING_FATAL_ERROR

Bug Check 0x11E: TOO_MANY_RECURSIVE_FAULTS

Bug Check 0x11F: INVALID_DRIVER_HANDLE

Bug Check 0x120: BITLOCKER_FATAL_ERROR

Bug Check 0x121: DRIVER_VIOLATION

Bug Check 0x122: WHEA_INTERNAL_ERROR

Bug Check 0x123: CRYPTO_SELF_TEST_FAILURE

Bug Check 0x124: WHEA_UNCORRECTABLE_ERROR

Bug Check 0x125: NMR_INVALID_STATE

Bug Check 0x126: NETIO_INVALID_POOL_CALLER

Bug Check 0x127: PAGE_NOT_ZERO

Bug Check 0x128: WORKER_THREAD_RETURNED_WITH_BAD_IO_PRIORITY

Bug Check 0x129: WORKER_THREAD_RETURNED_WITH_BAD_PAGING_IO_PRIORITY

Bug Check 0x12A: MUI_NO_VALID_SYSTEM_LANGUAGE

Bug Check 0x12B: FAULTY_HARDWARE_CORRUPTED_PAGE

Bug Check 0x12C: EXFAT_FILE_SYSTEM

Bug Check 0x12D: VOLSNAP_OVERLAPPED_TABLE_ACCESS

Bug Check 0x12E: INVALID_MDL_RANGE

Bug Check 0x12F: VHD_BOOT_INITIALIZATION_FAILED

Bug Check 0x130: DYNAMIC_ADD_PROCESSOR_MISMATCH

Bug Check 0x131: INVALID_EXTENDED_PROCESSOR_STATE

Bug Check 0x132: RESOURCE_OWNER_POINTER_INVALID

Bug Check 0x133 DPC_WATCHDOG_VIOLATION

Bug Check 0x134: DRIVE_EXTENDER

Bug Check 0x135: REGISTRY_FILTER_DRIVER_EXCEPTION

Bug Check 0x136: VHD_BOOT_HOST_VOLUME_NOT_ENOUGH_SPACE

Bug Check 0x137: WIN32K_HANDLE_MANAGER

Bug Check 0x138: GPIO_CONTROLLER_DRIVER_ERROR

Bug Check 0x139: KERNEL_SECURITY_CHECK_FAILURE

Bug Check 0x13A: KERNEL_MODE_HEAP_CORRUPTION

Bug Check 0x13B: PASSIVE_INTERRUPT_ERROR

Bug Check 0x13C: INVALID_IO_BOOST_STATE

Bug Check 0x13D: CRITICAL_INITIALIZATION_FAILURE

Bug Check 0x140: STORAGE_DEVICE_ABNORMALITY_DETECTED

Bug Check 0x141: VIDEO_ENGINE_TIMEOUT_DETECTED

Bug Check 0x142: VIDEO_TDR_APPLICATION_BLOCKED

Bug Check 0x143: PROCESSOR_DRIVER_INTERNAL

Bug Check 0x144: BUGCODE_USB3_DRIVER

Bug Check 0x145: SECURE_BOOT_VIOLATION

Bug Check 0x147: ABNORMAL_RESET_DETECTED

Bug Check 0x14B: SOC_SUBSYSTEM_FAILURE

Bug Check 0x149: REFS_FILE_SYSTEM

Bug Check 0x14A: KERNEL_WMI_INTERNAL

Bug Check 0x14C: FATAL_ABNORMAL_RESET_ERROR

Bug Check 0x14D: EXCEPTION_SCOPE_INVALID

Bug Check 0x14E: SOC_CRITICAL_DEVICE_REMOVED

Bug Check 0x14F: PDC_WATCHDOG_TIMEOUT

Bug Check 0x150: TCPIP_AOAC_NIC_ACTIVE_REFERENCE_LEAK

Bug Check 0x151: UNSUPPORTED_INSTRUCTION_MODE

Bug Check 0x152: INVALID_PUSH_LOCK_FLAGS

Bug Check 0x153: KERNEL_LOCK_ENTRY_LEAKED_ON_THREAD_TERMINATION

Bug Check 0x154: UNEXPECTED_STORE_EXCEPTION

Bug Check 0x155: OS_DATA_TAMPERING

Bug Check 0x156: WINSOCK_DETECTED_HUNG_CLOSESOCKET_LIVEDUMP

Bug Check 0x157: KERNEL_THREAD_PRIORITY_FLOOR_VIOLATION

Bug Check 0x158: ILLEGAL_IOMMU_PAGE_FAULT

Bug Check 0x159: HAL_ILLEGAL_IOMMU_PAGE_FAULT

Bug Check 0x15A: SDBUS_INTERNAL_ERROR

Bug Check 0x15B:
WORKER_THREAD_RETURNED_WITH_SYSTEM_PAGE_PRIORITY_ACTIVE

Bug Check 0x15C: PDC_WATCHDOG_TIMEOUT_LIVEDUMP

Bug Check 0x15D: SOC_SUBSYSTEM_FAILURE_LIVEDUMP

Bug Check 0x15E: BUGCODE_NDIS_DRIVER_LIVE_DUMP

Bug Check 0x15F: CONNECTED_STANDBY_WATCHDOG_TIMEOUT_LIVEDUMP

Bug Check 0x160: WIN32K_ATOMIC_CHECK_FAILURE

Bug Check 0x161: LIVE_SYSTEM_DUMP

Bug Check 0x162: KERNEL_AUTO_BOOST_INVALID_LOCK_RELEASE

Bug Check 0x163: WORKER_THREAD_TEST_CONDITION

Bug Check 0x164: WIN32K_CRITICAL_FAILURE

Bug Check 0x165: CLUSTER_CSV_STATUS_IO_TIMEOUT_LIVEDUMP

Bug Check 0x166: CLUSTER_RESOURCE_CALL_TIMEOUT_LIVEDUMP

Bug Check 0x167: CLUSTER_CSV_SNAPSHOT_DEVICE_INFO_TIMEOUT_LIVEDUMP

Bug Check 0x168: CLUSTER_CSV_STATE_TRANSITION_TIMEOUT_LIVEDUMP

Bug Check 0x169: CLUSTER_CSV_VOLUME_ARRIVAL_LIVEDUMP

Bug Check 0x16A: CLUSTER_CSV_VOLUME_REMOVAL_LIVEDUMP

Bug Check 0x16B: CLUSTER_CSV_CLUSTER_WATCHDOG_LIVEDUMP

Bug Check 0x16C: INVALID_RUNDOWN_PROTECTION_FLAGS

Bug Check 0x16D: INVALID_SLOT_ALLOCATOR_FLAGS

Bug Check 0x16E: ERESOURCE_INVALID_RELEASE

Bug Check 0x16F: CLUSTER_CSV_STATE_TRANSITION_INTERVAL_TIMEOUT_LIVEDUMP

Bug Check 0x170: CRYPTO_LIBRARY_INTERNAL_ERROR

Bug Check 0x171: CLUSTER_CSV_CLUSSVC_DISCONNECT_WATCHDOG

Bug Check 0x173: COREMSGCALL_INTERNAL_ERROR

Bug Check 0x174: COREMSG_INTERNAL_ERROR

Bug Check 0x175: PREVIOUS_FATAL_ABNORMAL_RESET_ERROR

Bug Check 0x178: ELAM_DRIVER_DETECTED_FATAL_ERROR

Bug Check 0x179: CLUSTER_CLUSPORT_STATUS_IO_TIMEOUT_LIVEDUMP

Bug Check 0x17B PROFILER_CONFIGURATION_ILLEGAL

Bug Check 0x17C PDC_LOCK_WATCHDOG_LIVEDUMP

Bug Check 0x17D PDC_UNEXPECTED_REVOCATION_LIVEDUMP

Bug Check 0x17E MICROCODE_REVISION_MISMATCH

Bug Check 0x187: VIDEO_DWMINIT_TIMEOUT_FALLBACK_BDD

Bug Check 0x188: CLUSTER_CSVFS_LIVEDUMP

Bug Check 0x189: BAD_OBJECT_HEADER

Bug Check 0x18B: SECURE_KERNEL_ERROR

Bug Check 0x18C: HYPERGUARD_VIOLATION

Bug Check 0x18D: SECURE_FAULT_UNHANDLED

Bug Check 0x18E: KERNEL_PARTITION_REFERENCE_VIOLATION

Bug Check 0x190: WIN32K_CRITICAL_FAILURE_LIVEDUMP

Bug Check 0x191: PF_DETECTED_CORRUPTION

Bug Check 0x192: KERNEL_AUTO_BOOST_LOCK_ACQUISITION_WITH_RAISED_IRQL

Bug Check 0x193: VIDEO_DXGKRNL_LIVEDUMP

Bug Check 0x195: SMB_SERVER_LIVEDUMP

Bug Check 0x196: LOADER_ROLLBACK_DETECTED

Bug Check 0x197: WIN32K_SECURITY_FAILURE

Bug Check 0x198: UFX_LIVEDUMP

Bug Check 0x199: KERNEL_STORAGE_SLOT_IN_USE

Bug Check 0x19A: WORKER_THREAD_RETURNED_WHILE_ATTACHED_TO_SILO

Bug Check 0x19B: TTM_FATAL_ERROR

Bug Check 0x19C: WIN32K_POWER_WATCHDOG_TIMEOUT

Bug Check 0x19D: CLUSTER_SVHDX_LIVEDUMP

Bug Check 0x1A0: TTM_WATCHDOG_TIMEOUT

Bug Check 0x1A3: CALL_HAS_NOT_RETURNED_WATCHDOG_TIMEOUT_LIVEDUMP

Bug Check 0x1A4: DRIPS_SW_HW_DIVERGENCE_LIVEDUMP

Bug Check 0x1A5: USB_DRIPS_BLOCKER_SURPRISE_REMOVAL_LIVEDUMP

Bug Check 0x1A6: BLUETOOTH_ERROR_RECOVERY_LIVEDUMP

Bug Check 0x1A7: SMB_REDIRECTOR_LIVEDUMP

Bug Check 0x1A8: VIDEO_DXGKRNL_BLACK_SCREEN_LIVEDUMP

Bug Check 0x1B0: VIDEO_MINIPORT_FAILED_LIVEDUMP

Bug Check 0x1C4: DRIVER_VERIFIER_DETECTED_VIOLATION_LIVEDUMP

Bug Check 0x1C5: IO_THREADPOOL_DEADLOCK_LIVEDUMP

Bug Check 0x1C6: FAST_ERESOURCE_PRECONDITION_VIOLATION

Bug Check 0x1C7: STORE_DATA_STRUCTURE_CORRUPTION

Bug Check 0x1C8 MANUALLY_INITIATED_POWER_BUTTON_HOLD

Bug Check 0x1C9: USER_MODE_HEALTH_MONITOR_LIVEDUMP

Bug Check 0x1CA: SYNTHETIC_WATCHDOG_TIMEOUT

Bug Check 0x1CB: INVALID_SILO_DETACH

Bug Check 0x1CC: EXRESOURCE_TIMEOUT_LIVEDUMP

Bug Check 0x1CD: INVALID_CALLBACK_STACK_ADDRESS

Bug Check 0x1CE: INVALID_KERNEL_STACK_ADDRESS

Bug Check 0x1CF: HARDWARE_WATCHDOG_TIMEOUT

Bug Check 0x1D0: CPI_FIRMWARE_WATCHDOG_TIMEOUT

Bug Check 0x1D1: TELEMETRY_ASSERTS_LIVEDUMP

Bug Check 0x1D2: WORKER_THREAD_INVALID_STATE

Bug Check 0x1D3: WFP_INVALID_OPERATION

Bug Check 0x1D4: UCMUCSI_LIVEDUMP

Bug Check 0x1D5: DRIVER_PNP_WATCHDOG

Bug Check 0x1D6: WORKER_THREAD_RETURNED_WITH_NON_DEFAULT_WORKLOAD_CLASS

Bug Check 0x1D7: EFS_FATAL_ERROR

Bug Check 0x1D8: UCMUCSI_FAILURE

Bug Check 0x1D9: HAL_IOMMU_INTERNAL_ERROR

Bug Check 0x1DB: IPI_WATCHDOG_TIMEOUT

Bug Check 0x1DC: DMA_COMMON_BUFFER_VECTOR_ERROR

Bug Check 0x356: XBOX_ERACTRL_CS_TIMEOUT

Bug Check 0xBFE: BC_BLUETOOTH_VERIFIER_FAULT

Bug Check 0xBFF: BC_BTHMINI_VERIFIER_FAULT

Bug Check 0x20001: HYPERVISOR_ERROR

Bug Check 0x1000007E: SYSTEM_THREAD_EXCEPTION_NOT_HANDLED_M

Bug Check 0x1000007F: UNEXPECTED_KERNEL_MODE_TRAP_M

Bug Check 0x1000008E: KERNEL_MODE_EXCEPTION_NOT_HANDLED_M

Bug Check 0x100000EA: THREAD_STUCK_IN_DEVICE_DRIVER_M

Bug Check 0x4000008A: THREAD_TERMINATE_HELD_MUTEX

Bug Check 0xC0000218: STATUS_CANNOT_LOAD_REGISTRY_FILE

Bug Check 0xC000021A: WINLOGON_FATAL_ERROR

Bug Check 0xC0000221: STATUS_IMAGE_CHECKSUM_MISMATCH

Bug Check 0xDEADDEAD: MANUALLY_INITIATED_CRASH1

- Debugger Reference
  - Debugger Reference
  - Command-Line Options
    - Command-Line Options
    - CDB Command-Line Options
    - KD Command-Line Options
    - WinDbg Command-Line Options
    - DbgSrv Command-Line Options
    - KdSrv Command-Line Options
    - DbEngPrx Command-Line Options
    - KDbgCtrl Command-Line Options
    - DbgRpc Command-Line Options
    - SymStore Command-Line Options
  - Environment Variables
    - Environment Variables
    - General Environment Variables
    - Kernel-Mode Environment Variables
  - Debugger Commands
    - Debugger Commands
    - Syntax Rules
      - Syntax Rules
      - Numerical Expression Syntax
        - Numerical Expression Syntax
        - MASM Numbers and Operators
        - C++ Numbers and Operators
        - MASM Expressions vs. C++ Expressions
        - Expression Examples
        - Sign Extension
      - String Wildcard Syntax
      - Register Syntax
      - Pseudo-Register Syntax
      - Source Line Syntax

Address and Address Range Syntax
Thread Syntax
Process Syntax
System Syntax
Multiprocessor Syntax
Command Tokens
Command Tokens
; (Command Separator)
{ } (Block Delimiter)
${ } (Alias Interpreter)
$$ (Comment Specifier)
* (Comment Line Specifier)
.block
.break
.catch
.continue
.do
.else
.elsif
.for
.foreach
.if
.leave
.printf
.while
Commands
Commands
ENTER (Repeat Last Command)
$<, $><, $$<, $$><, $$ >a< (Run Script File)
? (Command Help)
? (Evaluate Expression)
?? (Evaluate C++ Expression)

(Search for Disassembly Pattern)

|| (System Status)

||s (Set Current System)

| (Process Status)

|s (Set Current Process)

~ (Thread Status)

~e (Thread-Specific Command)

~f (Freeze Thread)

~u (Unfreeze Thread)

~n (Suspend Thread)

~m (Resume Thread)

~s (Set Current Thread)

~s (Change Current Processor)

a (Assemble)

ad (Delete Alias)

ah (Assertion Handling)

al (List Aliases)

as, aS (Set Alias)

ba (Break on Access)

bc (Breakpoint Clear)

bd (Breakpoint Disable)

be (Breakpoint Enable)

bl (Breakpoint List)

bp, bu, bm (Set Breakpoint)

br (Breakpoint Renumber)

bs (Update Breakpoint Command)

bsc (Update Conditional Breakpoint)

c (Compare Memory)

d, da, db, dc, dd, dD, df, dp, dq, du, dw, dW, dyb, dyd (Display Memory)

dda, ddp, ddu, dpa, dpp, dpu, dqa, dqp, dqu (Display Referenced Memory)

dds, dps, dqs (Display Words and Symbols)

dg (Display Selector)

dl (Display Linked List)

ds, dS (Display String)

dt (Display Type)

dtx (Display Type - Extended Debugger Object Model Information)

dv (Display Local Variables)

dx (Display Debugger Object Model Expression)

e, ea, eb, ed, eD, ef, ep, eq, eu, ew, eza, ezu (Enter Values)

f, fp (Fill Memory)

g (Go)

gc (Go from Conditional Breakpoint)

gh (Go with Exception Handled)

gn, gN (Go with Exception Not Handled)

gu (Go Up)

ib, iw, id (Input from Port)

j (Execute If - Else)

k, kb, kc, kd, kp, kP, kv (Display Stack Backtrace)

l+, l- (Set Source Options)

ld (Load Symbols)

lm (List Loaded Modules)

ln (List Nearest Symbols)

ls, lsa (List Source Lines)

lsc (List Current Source)

lse (Launch Source Editor)

lsf, lsf- (Load or Unload Source File)

lsp (Set Number of Source Lines)

m (Move Memory)

n (Set Number Base)

ob, ow, od (Output to Port)

p (Step)

pa (Step to Address)

pc (Step to Next Call)

pct (Step to Next Call or Return)

ph (Step to Next Branching Instruction)
pt (Step to Next Return)
q, qq (Quit)
qd (Quit and Detach)
r (Registers)
rdmsr (Read MSR)
rm (Register Mask)
s (Search Memory)
so (Set Kernel Debugging Options)
sq (Set Quiet Mode)
ss (Set Symbol Suffix)
sx, sxd, sxe, sxi, sxn, sxr, sx- (Set Exceptions)
t (Trace)
ta (Trace to Address)
tb (Trace to Next Branch)
tc (Trace to Next Call)
tct (Trace to Next Call or Return)
th (Trace to Next Branching Instruction)
tt (Trace to Next Return)
u (Unassemble)
uf (Unassemble Function)
up (Unassemble from Physical Memory)
ur (Unassemble Real Mode BIOS)
ux (Unassemble x86 BIOS)
vercommand (Show Debugger Command Line)
version (Show Debugger Version)
vertarget (Show Target Computer Version)
wrmsr (Write MSR)
wt (Trace and Watch Data)
x (Examine Symbols)
z (Execute While)
Meta-Commands

Meta-Commands

.abandon (Abandon Process)

.allow_exec_cmds (Allow Execution Commands)

.allow_image_mapping (Allow Image Mapping)

.apply_dbp (Apply Data Breakpoint to Context)

.asm (Change Disassembly Options)

.attach (Attach to Process)

.beep (Speaker Beep)

.bpcmds (Display Breakpoint Commands)

.bpsync (Synchronize Threads at Breakpoint)

.breakin (Break to the Kernel Debugger)

.browse (Display Command in Browser)

.bugcheck (Display Bug Check Data)

.cache (Set Cache Size)

.call (Call Function)

.chain (List Debugger Extensions)

.childdbg (Debug Child Processes)

.clients (List Debugging Clients)

.closehandle (Close Handle)

.cls (Clear Screen)

.context (Set User-Mode Address Context)

.copysym (Copy Symbol Files)

.cordll (Control CLR Debugging)

.crash (Force System Crash)

.create (Create Process)

.createdir (Set Created Process Directory)

.cxr (Display Context Record)

.dbgdbg (Debug Current Debugger)

.detach (Detach from Process)

.dml_flow (Unassemble with Links)

.dml_start (Display DML Starting Point)

.dump (Create Dump File)

.dumpcab (Create Dump File CAB)

.dvalloc (Allocate Memory)

.dvfree (Free Memory)

.echo (Echo Comment)

.echocpunum (Show CPU Number)

.echotime (Show Current Time)

.echotimestamps (Show Time Stamps)

.ecxr (Display Exception Context Record)

.effmach (Effective Machine)

.enable_long_status (Enable Long Integer Display)

.enable_unicode (Enable Unicode Display)

.endpsrv (End Process Server)

.endsrv (End Debugging Server)

.enumtag (Enumerate Secondary Callback Data)

.event_code (Display Event Code)

.eventlog (Display Recent Events)

.exepath (Set Executable Path)

.expr (Choose Expression Evaluator)

.exptr (Display Exception Pointers)

.exr (Display Exception Record)

.excr (Display Exception Context Record)

.extmatch (Display All Matching Extensions)

.extpath (Set Extension Path)

.f+, .f- (Shift Local Context)

.fiber (Set Fiber Context)

.fiximports (Fix Target Module Imports)

.flash_on_break (Flash on Break)

.fnent (Display Function Data)

.fnret (Display Function Return Value)

.force_radix_output (Use Radix for Integers)

.force_tb (Forcibly Allow Branch Tracing)

.formats (Show Number Formats)

.fpo (Control FPO Overrides)

.frame (Set Local Context)

.help (Meta-Command Help)

.hh (Open HTML Help File)

.hideinjectedcode (Hide Injected Code)

.holdmem (Hold and Compare Memory)

.idle_cmd (Set Idle Command)

.ignore_missing_pages (Suppress Missing Page Errors)

.inline (Toggle Inline Function Debugging)

.imgscan (Find Image Headers)

.jdinfo (Use JIT_DEBUG_INFO)

.kdfiles (Set Driver Replacement Map)

.kdtargetmac (Display Target MAC Address)

.kframes (Set Stack Length)

.kill (Kill Process)

.lastevent (Display Last Event)

.lines (Toggle Source Line Support)

.load, .loadby (Load Extension DLL)

.locale (Set Locale)

.logappend (Append Log File)

.logclose (Close Log File)

.logfile (Display Log File Status)

.logopen (Open Log File)

.netsyms (Disable Network Symbol Loading)

.netuse (Control Network Connections)

.noshell (Prohibit Shell Commands)

.noversion (Disable Version Checking)

.ocommand (Expect Commands from Target)

.nvload (NatVis Load)

.nvlist (NatVis List)

.nvunload (NatVis Unload)

.nvunloadall (NatVis Unload All)

.ofilter (Filter Target Output)

.open (Open Source File)

.opendump (Open Dump File)

.outmask (Control Output Mask)

.pagein (Page In Memory)

.pcmd (Set Prompt Command)

.pop (Restore Debugger State)

.prefer_dml (Prefer Debugger Markup Language)

.process (Set Process Context)

.prompt_allow (Control Prompt Display)

.push (Save Debugger State)

.quit_lock (Prevent Accidental Quit)

.readmem (Read Memory from File)

.reboot (Reboot Target Computer)

.record_branches (Enable Branch Recording)

.reload (Reload Module)

.remote (Create Remote.exe Server)

.remote_exit (Exit Debugging Client)

.restart (Restart Target Application)

.restart (Restart Kernel Connection)

.rrestart (Register for Restart)

.scroll_prefs (Control Source Scrolling Preferences)

.scriptdebug (Debug JavaScript)

.scriptlist (List Loaded Scripts)

.scriptload (Load Script)

.scriptproviders (List Script Providers)

.scriptrun (Run Script)

.scriptunload (Unload Script)

.secure (Activate Secure Mode)

.send_file (Send File)

.server (Create Debugging Server)

.servers (List Debugging Servers)

.setdll (Set Default Extension DLL)
.shell (Command Shell)
.settings (Set Debug Settings)
.show_read_failures
.show_sym_failures
.sleep (Pause Debugger)
.sound_notify (Use Notification Sound)
.srcfix, .lsrcfix (Use Source Server)
.srcnoisy (Noisy Source Loading)
.srcpath, .lsrcpath (Set Source Path)
.step_filter (Set Step Filter)
.suspend_ui (Suspend WinDbg Interface)
.symfix (Set Symbol Store Path)
.symopt (Set Symbol Options)
.sympath (Set Symbol Path)
.thread (Set Register Context)
.time (Display System Time)
.tlist (List Process IDs)
.trap (Display Trap Frame)
.tss (Display Task State Segment)
.ttime (Display Thread Times)
.typeopt (Set Type Options)
.unload (Unload Extension DLL)
.unloadall (Unload All Extension DLLs)
.urestart (Unregister for Restart)
.wake (Wake Debugger)
.write_cmd_hist (Write Command History)
.writemem (Write Memory to File)
.wtitle (Set Window Title)
Control Keys
Control Keys
CTRL+\ (Debug Current Debugger)

CTRL+ALT+\ (Debug Current Debugger)
CTRL+A (Toggle Baud Rate)
CTRL+B (Quit Local Debugger)
CTRL+C (Break)
CTRL+D (Toggle Debug Info)
CTRL+F (Break to KD)
CTRL+K (Change Post-Reboot Break State)
CTRL+P (Debug Current Debugger)
CTRL+R (Re-synchronize)
CTRL+V (Toggle Verbose Mode)
CTRL+W (Show Debugger Version)
General Extension Commands
General Extension Commands
!acl
!address
!analyze
!asd
!atom
!bitcount
!blackboxbsd
!blackboxscm
!chksym
!chkimg
!cppexr
!cpuid
!cs
!cxr
!dh
!dlls
!dml_proc
!dumpfa
!elog_str

!envvar

!error

!exchain

!exr

!findxmldata

!for_each_frame

!for_each_function

!for_each_local

!for_each_module

!for_each_register

!gflag

!gle

!gs

!handle

!heap

!help

!homedir

!hstring

!hstring2

!htrace

!imggp

!imgreloc

!kuser

!list

!lmi

!mui

!net_send

!obja

!owner

!peb

!rebase

!rtlavl

!sd
!sid
!slist
!std_map
!stl
!str
!sym
!symsrv
!teb
!tls
!token
!tp
!triage
!ustr
!version
!winrterr
Kernel-Mode Extension Commands
Kernel-Mode Extension Commands
!ahcache
!alignmentfaults
!analyzebugcheck
!apc
!apicerr
!arbinst
!arbiter
!ate
!bcb
!blockeddrv
!bpid
!btb
!bth
!bugdump

!bushnd

!ca

!callback

!calldata

!can_write_kdump

!cbreg

!cchelp

!chklowmem

!cmreslist

!cpuinfo

!db, !dc, !dd, !dp, !dq, !du, !dw

!dbgprint

!dblink

!dcr

!dcs

!deadlock

!defwrites

!devext

!devhandles

!devnode

!devobj

!devstack

!dflink

!diskspace

!dma

!dpa

!dpcs

!driveinfo

!drivers

!drvobj

!dskheap

!eb, !ed

!ecb, !ecd, !ecw

!ecs

!errlog

!errpkt

!errrec

!exca

!filecache

!filelock

!fileobj

!filetime

!finddata

!findfilelockowner

!findthreads

!for_each_process

!for_each_thread

!fpsearch

!frozen

!fwver

!fxdevice

!gbl

!gentable

!hidppd

!ib, !id, !iw

!icpleak

!idt

!ih

!ihs

!ioresdes

!ioctldecode

!ioreslist

!iovirp

!ipi

!irp

!irpfind

!irpzone

!irql

!isainfo

!isr

!ivt

!job

!kb, !kv

!loadermemorylist

!lockedpages

!locks (!kdext*.locks)

!logonsession

!lookaside

!lpc

!mca

!memlist

!memusage

!mps

!mtrr

!npx

!ob, !od, !ow

!object

!obtrace

!openmaps

!pars

!pat

!pci

!pciir

!pcitree

!pcm

!pcr

!pcrs

!pfn

!pmc

!pmssa

!powertriage

!pnpevent

!pocaps

!pool

!poolfind

!poolused

!poolval

!popolicy

!pplookaside

!ppmidle

!ppmidleaccounting

!ppmidlepolicy

!ppmlpscenarios

!ppmperf

!ppmperfpolicy

!ppmsettings

!ppmstate

!prcb

!process

!processfields

!processirps

!psp

!pte

!pte2va

!ptov

!qlocks

!ready

!reg

!regkcb

!rellist

!ruleinfo

!running

!scm

!search

!searchpte

!sel

!session

!smt

!sprocess

!srb

!stacks

!swd

!sysinfo

!sysptes

!thread

!threadfields

!time

!timer

!tokenfields

!trap

!tss

!tz

!tzinfo

!ubc

!ubd

!ube

!ubl

!ubp

!urb

!vad

!vad_reload
!validatelist
!verifier
!vm
!vpb
!vpdd
!vtop
!wdmaud
!whattime
!whatperftime
!whea
!wsle
!zombies
User-Mode Extension Commands
User-Mode Extension Commands
!avrf
!critsec
!dp (!ntsdexts.dp)
!dreg
!dt
!findstack
!gatom
!igrep
!locks (!ntsdexts.locks)
!mapped_file
!runaway
!threadtoken
!uniqstack
!vadump
!vprot
Specialized Extension Commands
Specialized Extension Commands

Storage Kernel Debugger Extensions
Storage Kernel Debugger Extensions
!storagekd.storadapter
!storagekd.storclass
!storagekd.storhelp
!storagekd.storlogirp
!storagekd.storloglist
!storagekd.storlogsrb
!storagekd.storsrb
!storagekd.storunit
Bluetooth Extensions (Bthkd.dll)
Bluetooth Extensions (Bthkd.dll)
!bthkd.bthdevinfo
!bthkd.bthenuminfo
!bthkd.bthinfo
!bthkd.bthhelp
!bthkd.bthtree
!bthkd.bthusbtransfer
!bthkd.dibflags
!bthkd.hcicmd
!bthkd.hciinterface
!bthkd.l2capinterface
!bthkd.rfcomminfo
!bthkd.rfcommconnection
!bthkd.rfcommchannel
!bthkd.sdpinterface
!bthkd.scointerface
!bthkd.sdpnode
!bthkd.sdpstream
GPIO Extensions
GPIO Extensions
!gpiokd.help

!gpiokd.bankinfo
!gpiokd.clientlist
!gpiokd.gpioext
!gpiokd.pininfo
!gpiokd.pinisrvec
!gpiokd.pintable
USB 3.0 Extensions
USB 3.0 Extensions
USB 3.0 Data Structures
!usb3kd.help
!usb3kd.device_info
!usb3kd.device_info_from_pdo
!usb3kd.dsf
!usb3kd.hub_info
!usb3kd.hub_info_from_fdo
!usb3kd.port_info
!usb3kd.ucx_device
!usb3kd.ucx_endpoint
!usb3kd.ucx_controller
!usb3kd.ucx_controller_list
!usb3kd.usbanalyze
!usb3kd.usbdstatus
!usb3kd.usb_tree
!usb3kd.urb
!usb3kd.xhci_capability
!usb3kd.xhci_commandring
!usb3kd.xhci_deviceslots
!usb3kd.xhci_dumpall
!usb3kd.xhci_eventring
!usb3kd.xhci_findowner
!usb3kd.xhci_info
!usb3kd.xhci_registers

!usb3kd.xhci_resourceusage
!usb3kd.xhci_trb
!usb3kd.xhci_transferring
USB 2.0 Debugger Extensions
USB 2.0 Debugger Extensions
!usbkd.usbhelp
!usbkd._ehcidd
!usbkd._ehciep
!usbkd._ehciframe
!usbkd._ehciqh
!usbkd._ehciregs
!usbkd._ehcisitd
!usbkd._ehcistq
!usbkd._ehcitd
!usbkd._ehcitfer
!usbkd._ehciitd
!usbkd.doesdumphaveusbdata
!usbkd.isthisdumpasyncissue
!usbkd.urbfunc
!usbkd.usb2
!usbkd.usb2tree
!usbkd.usbchain
!usbkd.usbdevobj
!usbkd.usbdpc
!usbkd.ehci_info_from_fdo
!usbkd.usbdevh
!usbkd.usbep
!usbkd.usbfaildata
!usbkd.usbhcdext
!usbkd.usbdstatus
!usbkd.usbhcdhccontext
!usbkd.usbhcdlist

!usbkd.usbhcdlistlogs
!usbkd.usbhcdlog
!usbkd.usbhcdlogex
!usbkd.usbhcdpnp
!usbkd.usbhcdpow
!usbkd.hub2_info_from_fdo
!usbkd.usbhuberr
!usbkd.usbhubext
!usbkd.usbhubinfo
!usbkd.usbhublog
!usbkd.usbhubmddevext
!usbkd.usbhubmdpd
!usbkd.usbhubpd
!usbkd.usbhubs
!usbkd.usblist
!usbkd.usbpo
!usbkd.usbpdos
!usbkd.usbpdoxls
!usbkd.usbpnp
!usbkd.usbportisasyncadv
!usbkd.usbportmdportlog
!usbkd.usbportmddcontext
!usbkd.usbportmddevext
!usbkd.usbtriage
!usbkd.usbtt
!usbkd.usbtx
!usbkd.usbusb2ep
!usbkd.usbusb2tt
!usbkd.usbver
RCDRKD Extensions
RCDRKD Extensions
!rcdrkd.rcdrhelp

!rcdrkd.rcdrcrashdump
!rcdrkd.rcdrlogdump
!rcdrkd.rcdrloglist
!rcdrkd.rcdrlogsave
!rcdrkd.rcdrsearchpath
!rcdrkd.rcdrsettraceprefix
!rcdrkd.rcdrtmffile
!rcdrkd.rcdrtraceprtdebug
HID Extensions
HID Extensions
!hidkd.help
!hidkd.hidfdo
!hidkd.hidpdo
!hidkd.hidtree
!hidkd.hidppd
!hidkd.hidrd
Logger Extensions (Logexts.dll)
Logger Extensions (Logexts.dll)
!logexts.help
!logexts.logb
!logexts.logc
!logexts.logd
!logexts.loge
!logexts.logi
!logexts.logm
!logexts.logo
NDIS Extensions (Ndiskd.dll)
NDIS Extensions (Ndiskd.dll)
!ndiskd.help
!ndiskd.netadapter
!ndiskd.minidriver
!ndiskd.rcvqueue

!ndiskd.protocol

!ndiskd.mopen

!ndiskd.filter

!ndiskd.filterdriver

!ndiskd.nbl

!ndiskd.nb

!ndiskd.nblpool

!ndiskd.nbpool

!ndiskd.pendingnbls

!ndiskd.nbllog

!ndiskd.cxadapter

!ndiskd.netqueue

!ndiskd.netrb

!ndiskd.netpacket

!ndiskd.netpacketfragment

!ndiskd.oid

!ndiskd.interfaces

!ndiskd.ifprovider

!ndiskd.ifstacktable

!ndiskd.compartments

!ndiskd.pkt

!ndiskd.pktpools

!ndiskd.findpacket

!ndiskd.vc

!ndiskd.af

!ndiskd.ndisref

!ndiskd.ndisevent

!ndiskd.ndisstack

!ndiskd.wdiadapter

!ndiskd.wdiminidriver

!ndiskd.nwadapter

!ndiskd.ndisrwlock

!ndiskd.ndisslot
!ndiskd.ndis
!ndiskd.dbglevel
!ndiskd.dbgsystems
!ndiskd.ndiskdversion
!ndiskd.netreport
RPC Extensions (Rpcexts.dll)
RPC Extensions (Rpcexts.dll)
!rpcexts.checkrpcsym
!rpcexts.eeinfo
!rpcexts.eerecord
!rpcexts.getcallinfo
!rpcexts.getclientcallinfo
!rpcexts.getdbgcell
!rpcexts.getendpointinfo
!rpcexts.getthreadinfo
!rpcexts.help
!rpcexts.rpcreadstack
!rpcexts.rpctime
!rpcexts.thread
ACPI Extensions (Acpikd.dll and Kdexts.dll)
ACPI Extensions (Acpikd.dll and Kdexts.dll)
!acpicache
!acpiinf
!acpiirqarb
!acpikd.help
!amli ?
!amli bc
!amli bd
!amli be
!amli bl
!amli bp

!amli cl
!amli debugger
!amli dh
!amli dl
!amli dns
!amli do
!amli ds
!amli find
!amli lc
!amli ln
!amli r
!amli set
!amli u
!facs
!fadt
!mapic
!nsobj
!nstree
!rsdt
Graphics Driver Extensions (Gdikdx.dll)
Graphics Driver Extensions (Gdikdx.dll)
!gdikdx.verifier
Kernel Streaming Extensions (Ks.dll)
Kernel Streaming Extensions (Ks.dll)
!ks.help
!ks.kshelp
!ks.pchelp
!ks.allstreams
!ks.automation
!ks.devhdr
!ks.dhdr
!ks.dump

!ks.dumpbag
!ks.dumpcircuit
!ks.dumplog
!ks.dumpqueue
!ks.enumdevobj
!ks.enumdrvobj
!ks.eval
!ks.findlive
!ks.forcedump
!ks.graph
!ks.libexts
!ks.objhdr
!ks.ohdr
!ks.pciaudio
!ks.pciks
!ks.shdr
!ks.topology
SCSI Miniport Extensions (Scsikd.dll and Minipkd.dll)
SCSI Miniport Extensions (Scsikd.dll and Minipkd.dll)
!scsikd.help
!scsikd.classext
!scsikd.scsiext
!scsikd.srbdata
!minipkd.help
!minipkd.adapter
!minipkd.adapters
!minipkd.exports
!minipkd.lun
!minipkd.portconfig
!minipkd.req
!minipkd.srb
Windows Driver Framework Extensions (Wdfkd.dll)

Windows Driver Framework Extensions (Wdfkd.dll)

!wdfkd.help

!wdfkd.wdfchildlist

!wdfkd.wdfcollection

!wdfkd.wdfcommonbuffer

!wdfkd.wdfcrashdump

!wdfkd.wdfdevext

!wdfkd.wdfdevice

!wdfkd.wdfdeviceinterrupts

!wdfkd.wdfdevicequeues

!wdfkd.wdfdmaenabler

!wdfkd.wdfdmaenablers

!wdfkd.wdfdmatransaction

!wdfkd.wdfdriverinfo

!wdfkd.wdfextendwatchdog

!wdfkd.wdffindobjects

!wdfkd.wdfforwardprogress

!wdfkd.wdfgetdriver

!wdfkd.wdfhandle

!wdfkd.wdfhelp

!wdfkd.wdfinterrupt

!wdfkd.wdfiotarget

!wdfkd.wdfldr

!wdfkd.wdflogdump

!wdfkd.wdflogsave

!wdfkd.wdfmemory

!wdfkd.wdfobject

!wdfkd.wdfopenhandles

!wdfkd.wdfpool

!wdfkd.wdfpooltracker

!wdfkd.wdfpoolusage

!wdfkd.wdfqueue

!wdfkd.wdfrequest
!wdfkd.wdfsearchpath
!wdfkd.wdfsettraceprefix
!wdfkd.wdfsetdriver
!wdfkd.wdfspinlock
!wdfkd.wdftagtracker
!wdfkd.wdftmffile
!wdfkd.wdftraceprtdebug
!wdfkd.wdfumdevstack
!wdfkd.wdfumdevstacks
!wdfkd.wdfumdownirp
!wdfkd.wdfumfile
!wdfkd.wdfumirp
!wdfkd.wdfumirps
!wdfkd_wdfumtriage
!wdfkd.wdfusbdevice
!wdfkd.wdfusbinterface
!wdfkd.wdfusbpipe
!wdfkd.wdfwmi
User-Mode Driver Framework Extensions (Wudfext.dll)
User-Mode Driver Framework Extensions (Wudfext.dll)
!wudfext.help
!wudfext.umdevstack
!wudfext.umdevstacks
!wudfext.umfile
!wudfext.umirp
!wudfext.umirps
!wudfext.wudfdevice
!wudfext.wudfdevicequeues
!wudfext.wudfdownkmirp
!wudfext.wudfdriverinfo
!wudfext.wudfdumpobjects

!wudfext.wudffile
!wudfext.wudffilehandletarget
!wudfext.wudfiotarget
!wudfext.wudfobject
!wudfext.wudfqueue
!wudfext.wudfrefhist
!wudfext.wudfrequest
!wudfext.wudfusbinterface
!wudfext.wudfusbpipe
!wudfext.wudfusbtarget
WMI Tracing Extensions (Wmitrace.dll)
WMI Tracing Extensions (Wmitrace.dll)
!wmitrace.disable
!wmitrace.dumpmini
!wmitrace.dumpminievent
!wmitrace.dynamicprint
!wmitrace.enable
!wmitrace.eventlogdump
!wmitrace.help
!wmitrace.logdump
!wmitrace.logger
!wmitrace.logsave
!wmitrace.searchpath
!wmitrace.setprefix
!wmitrace.start
!wmitrace.stop
!wmitrace.strdump
!wmitrace.tfp
!wmitrace.tmffile
!wmitrace.traceoperation
OEM Support Extensions (kdex2x86.dll)
SieExtPub.dll

Debugger-Related APIs
Debugger-Related APIs
Symbol Server API
The dbgeng.h Header File
The wdbgexts.h Header File
Debugger Error and Warning Messages
Debugger Error and Warning Messages
dbgerr001: PEB is Paged Out
dbgerr002: Bad Pointer
dbgerr003: Mismatched Symbols
dbgerr004: Page Not Present in Dump File
dbgerr005: Private Symbols Required
Stack Unwind Information Not Available
No Header Information Available
WinDbg Graphical Interface Features
WinDbg Graphical Interface Features
File Menu
File Menu
File | Open Source File
File | Close Current Window
File | Open Executable
File | Attach to a Process
File | Open Crash Dump
File | Connect to Remote Session
File | Connect to Remote Stub
File | Kernel Debug
File | Symbol File Path
File | Source File Path
File | Image File Path
File | Open Workspace
File | Save Workspace
File | Save Workspace As

File | Clear Workspace
File | Delete Workspaces
File | Open Workspace in File
File | Save Workspace to File
File | Map Network Drive
File | Disconnect Network Drive
File | Recent Files
File | Exit
Edit Menu
Edit Menu
Edit | Cut
Edit | Copy
Edit | Paste
Edit | Select All
Edit | Write Window Text to File
Edit | Add to Command Output
Edit | Clear Command Output
Edit | Evaluate Selection
Edit | Display Selected Type
Edit | Find
Edit | Find Next
Edit | Go to Address
Edit | Go to Line
Edit | Go to Current Instruction
Edit | Set Current Instruction
Edit | Breakpoints
Edit | Open/Close Log File
View Menu
View Menu
View | Command
View | Watch
View | Locals

View | Registers
View | Memory
View | Call Stack
View | Disassembly
View | Scratch Pad
View | Processes and Threads
View | Command Browser
View | Verbose Output
View | Show Version
View | Toolbar
View | Status Bar
View | Font
View | Options
View | Source language file extensions
View | WinDbg Command Line
Debug Menu
Debug Menu
Debug | Go
Debug | Go Unhandled Exception
Debug | Go Handled Exception
Debug | Restart
Debug | Stop Debugging
Debug | Detach Debuggee
Debug | Break
Debug | Step Into
Debug | Step Over
Debug | Step Out
Debug | Run to Cursor
Debug | Source Mode
Debug | Resolve Unqualified Symbols
Debug | Event Filters
Debug | Modules

- Debug | Kernel Connection | Cycle Baud Rate
- Debug | Kernel Connection | Cycle Initial Break
- Debug | Kernel Connection | Resynchronize

Window Menu

- Window Menu
- Window | Close All Source Windows
- Window | Close All Error Windows
- Window | Open Dock
- Window | Dock All
- Window | Undock All
- Window | MDI Emulation
- Window | Automatically Open Disassembly
- List of Open Windows

Help Menu

- Help Menu
- Help | Contents
- Help | Index
- Help | Search
- Help | About

Toolbar Buttons

Keyboard Shortcuts

Debugging Tools for Windows (WinDbg, KD, CDB, NTSD)

8/5/2019 • 3 minutes to read • Edit Online

Start here for an overview of Debugging Tools for Windows. This tool set includes WinDbg and other debuggers.

Install Debugging Tools for Windows

You can get Debugging Tools for Windows as part of a development kit or as a standalone tool set:

- **As part of the WDK**

  Debugging Tools for Windows is included in the Windows Driver Kit (WDK). To get the WDK, see Download the Windows Driver Kit (WDK).

- **As part of the Windows SDK**

  Debugging Tools for Windows is included in the Windows Software Development Kit (SDK). To download the installer or an ISO image, see Windows 10 SDK on Windows Dev Center.

- **As a standalone tool set**

  You can install the Debugging Tools for Windows alone, without the Windows SDK or WDK, by starting installation of the Windows SDK and then selecting only **Debugging Tools for Windows** in the list of features to install (and clearing the selection of all other features). To download the installer or an ISO image, see Windows 10 SDK on Windows Dev Center.

Get started with Windows Debugging

To get started with Windows debugging, see Getting Started with Windows Debugging.

To get started with debugging kernel-mode drivers, see Debug Universal Drivers - Step by Step Lab (Echo Kernel-Mode). This is a step-by-step lab that shows how to use WinDbg to debug Echo, a sample driver that uses the Kernel-Mode Driver Framework (KMDF).

Debugging environments

If your computer has Visual Studio and the WDK installed, then you have six available debugging environments. For descriptions of these environments, see Debugging Environments.

All of these debugging environments provide user interfaces for the same underlying debugging engine, which is implemented in the Windows Symbolic Debugger Engine (Dbgeng.dll). This debugging engine is also called the *Windows debugger*, and the six debugging environments are collectively called the *Windows debuggers*.

> **NOTE**
>
> Visual Studio includes its own debugging environment and debugging engine, which together are called the *Visual Studio debugger*. For information on debugging in Visual Studio, see Debugging in Visual Studio. For debugging managed code, such as C#, using the Visual Studio debugger is often the easiest way to get started.

Windows debuggers

The Windows debuggers can run on x86-based, x64-based, or ARM-based processors, and they can debug code that is running on those same architectures. Sometimes the debugger and the code being debugged run on the same computer, but other times the debugger and the code being debugged run on separate computers. In either case, the computer that is running the debugger is called the *host computer*, and the computer that is being debugged is called the *target computer*. The Windows debuggers support the following versions of Windows for both the host and target computers.

- Windows 10 and Windows Server 2016
- Windows 8.1 and Windows Server 2012 R2
- Windows 8 and Windows Server 2012
- Windows 7 and Windows Server 2008 R2

Symbols and symbol files

Symbol files store a variety of data that are not required when running the executable binaries, but symbol files are very useful when debugging code. For more information about creating and using symbol files, see Symbols for Windows debugging (WinDbg, KD, CDB, NTSD).

Blue screens and crash dump files

If Windows stops working and displays a blue screen, the computer has shut down abruptly to protect itself from data loss and displays a bug check code. For more information, see Bug Checks (Blue Screens). You analyze crash dump files that are created when Windows shuts down by using WinDbg and other Windows debuggers. For more information, see Crash dump analysis using the Windows debuggers (WinDbg).

Tools and utilities

In addition to the debuggers, Debugging Tools for Windows includes a set of tools that are useful for debugging. For a full list of the tools, see Tools Included in Debugging Tools for Windows.

Additional documentation

For additional information related to Debugging Tools for Windows, see Debugging Resources. For information on what's new in Windows 10, see Debugging Tools for Windows: New for Windows 10.

Download Debugging Tools for Windows

10/7/2019 • 2 minutes to read • Edit Online

The Windows Debugger (WinDbg) can be used to debug kernel-mode and user-mode code, to analyze crash dumps, and to examine the CPU registers while the code executes.

Download WinDbg Preview

WinDbg Preview is a new version of WinDbg with more modern visuals, faster windows, a full-fledged scripting experience, built with the extensible debugger data model front and center. WinDbg Preview is using the same underlying engine as WinDbg today, so all the commands, extensions, and workflows still work as they did before.

- Download WinDbg Preview from the Microsoft Store: WinDbg Preview.
- Learn more about installation and configuration in WinDbg Preview - Installation.

Debugging Tools for Windows 10 (WinDbg)

If you just need the Debugging Tools for Windows 10, and not Windows Driver Kit (WDK) for Windows 10 or Visual Studio 2017, you can install the debugging tools as a standalone component from the Windows SDK. In the SDK installation wizard, select **Debugging Tools for Windows**, and deselect all other components.

- Get Debugging Tools for Windows (WinDbg) from the SDK: Windows 10 SDK.
- Learn more about WinDbg and other debuggers in Debugging Tools for Windows (WinDbg, KD, CDB, NTSD).

> **TIP**
> If the Windows SDK is already installed, open **Settings**, navigate to **Apps & features**, select **Windows Software Development Kit**, and then click **Modify** to change the installation to add **Debugging Tools for Windows**.

Looking for the debugging tools for earlier version of Windows?

To download the debugger tools for previous versions of Windows, you need to download the Windows SDK for the version you are debugging from the Windows SDK and emulator archive. In the installation wizard of the SDK, select **Debugging Tools for Windows**, and deselect all other components.

Looking for related downloads?

- Windows Driver Kit (WDK)
- Windows Debugger Symbols
- Windows HLK, HCK, or Logo Kit
- Windows Assessment and Deployment Kit (Windows ADK)
- Windows Insider Preview builds

Windows Symbol Packages

10/7/2019 • 2 minutes to read • Edit Online

Symbol files make it easier to debug your code. The easiest way to get Windows symbols is to use the Microsoft public symbol server. The symbol server makes symbols available to your debugging tools as needed. After a symbol file is downloaded from the symbol server it is cached on the local computer for quick access.

Symbol package deprecation

IMPORTANT

We are no longer publishing the offline symbol packages for Windows.

With the cadence that we release updates for Windows, the Windows debugging symbols we publish via the packages on this page are quickly made out of date. We have made significant improvements to the online Microsoft Symbol Server by moving this to be an Azure-based symbol store, and symbols for all Windows versions and updates are available there. You can find more about this in this blog entry.

For information on how to retrieve symbols for a machine that is not connected to the Internet, see Using a Manifest File with SymChk.

Symbol Resources and Feedback

To learn more about using symbols and debugging, see Symbols and Symbol Files.

For help with debugging issues, see Debugging Resources.

We are interested in your feedback about symbols. Please mail suggestions or bug reports to windbgfb@microsoft.com. Technical support is not available from this address, but your feedback will help us to plan future changes for symbols and will make them more useful to you in the future.

Looking for related downloads?

- Download Windows Debugging Tools
- Download the Windows Driver Kit (WDK)
- Download the Windows Assessment and Deployment Kit (Windows ADK)
- Download the Windows HLK, HCK, or Logo Kit
- Download Windows Insider Preview builds

Getting Started with Windows Debugging

9/9/2019 • 5 minutes to read • Edit Online

This article covers how to get started with Windows Debugging. If your goal is to use the debugger to analyze a crash dump, see Analyze crash dump files by using WinDbg.

To get started with Windows Debugging, complete the tasks that are described in this article.

1. Determine the host and the target

The debugger runs on the *host* system, and the code that you want to debug runs on the *target* system.

Host <----------> Target

Because it is common to stop instruction execution on the processor during debugging, two computer systems are typically used. In some situations, you might be able to use a virtual machine as the second system. For example, you might be able to use a virtual PC that is running on the same PC as the code that you need to debug. However, if your code is communicating to low-level hardware, using a virtual PC may not be the best approach. For more information, see Setting up network debugging of a virtual machine - KDNET.

2. Determine the type: kernel-mode or user-mode

Next, you need to determine whether you will do kernel-mode or user-mode debugging.

Kernel mode is the processor-access mode in which the operating system and privileged programs run. Kernel-mode code has permission to access any part of the system, and it is not restricted like user-mode code. Kernel-mode code can gain access to any part of any other process running in either user mode or kernel mode. Much of the core OS functionality and many hardware device drivers run in kernel mode.

User mode is the mode that applications and subsystems on the computer run in. Processes that run in user mode do so within their own virtual address spaces. They are restricted from gaining direct access to many parts of the system, including system hardware, memory that was not allocated for their use, and other portions of the system that might compromise system integrity. Because processes that run in user mode are effectively isolated from the system and other user-mode processes, they cannot interfere with these resources.

If your goal is to debug a driver, determine if the driver is a kernel-mode driver or a user-mode driver. Windows Driver Model (WDM) drivers and Kernel-Mode Driver Framework (KMDF) are both kernel-mode drivers. As the name sugests, User-Mode Driver Framework (UMDF) drivers are user-mode drivers.

For some issues, it can be difficult to determine which mode the code executes in. In that case, you may need to pick one mode and look to see what information is available in that mode. Some issues require using the debugger in both user mode and kernel mode.

Depending on what mode you decide to debug in, you will need to configure and use the debuggers in different ways. Some debugging commands operate the same in both modes, and some commands operate differently in different modes.

For information about using the debugger in kernel mode, see the following articles:

- Getting started with WinDbg (kernel-mode)
- Debug universal drivers - step by step lab (echo kernel-mode)
- Debug drivers - step by step lab (Sysvad kernel-mode).

For information about using the debugger in user mode, see Getting started with WinDbg (user-mode).

3. Choose your debugger environment

WinDbg works well in most situations, but there are times when you may want to use another debugger, such as console debuggers for automation or Visual Studio. For more information, see Debugging environments.

4. Determine how to connect the target and host

Typically, target and host systems are connected by an Ethernet network. If you are doing early bring-up work, or you don't have an Ethernet connection on a device, other network connection options are available. For more information, see these articles:

- Setting up KDNET network kernel debugging automatically
- Setting up kernel-mode debugging
- Setting up KDNET network kernel debugging manually
- Setting up network debugging of a virtual machine - KDNET

5. Choose either the 32-bit or 64-bit debugging tools

Which debugging tools to choose—32-bit or 64-bit—depends on the version of Windows that is running on the target and host systems and on whether you are debugging 32-bit or 64-bit code. For more information, see Choosing the 32-Bit or 64-Bit debugging tools.

6. Configure symbols

To use all of the advanced functionality that WinDbg provides, you must load the proper symbols. If you do not have symbols properly configured, you will receive messages indicating that symbols are not available when you attempt to use functionality that is dependent on symbols. For more information, see Symbols for Windows debugging (WinDbg, KD, CDB, NTSD).

7. Configure source code

If your goal is to debug your own source code, you will need to configure a path to your source code. For more information, see Source path.

8. Become familiar with debugger operation

The Debugger operation section of this documentation describes debugger operation for various tasks. For example, Loading debugger extension DLLs explains how to load debugger extensions. To learn more about working with WinDbg, see Debugging using WinDbg.

9. Become familiar with debugging techniques

Standard debugging techniques apply to most debugging scenarios, and examples include setting breakpoints, inspecting the call stack, and finding a memory leak. Specialized debugging techniques apply to particular technologies or types of code. Examples include Plug and Play debugging, KMDF debugging, and RPC debugging.

10. Use the debugger reference commands

Over time, you will use different debugging commands as you work in the debugger. Use the **.hh** (open HTML help file) command in the debugger to display help information about any debugging command. For more information about the available commands, see Debugger reference.

11. Use debugging extensions for specific technologies

There are multiple debugging extensions that provide parsing of domain-specific data structures. For more information, see Specialized extensions.

12. Learn about related Windows internals

This documentation assumes a knowledge of Windows internals. To learn more about Windows internals (including memory usage, context, threads, and processes), review additional resources, such as *Windows Internals* by Mark Russinovich, David Solomon, and Alex Ionescu.

13. Review additional debugging resources

Additional resources include the following books and videos:

- *Inside Windows Debugging: Practical Debugging and Tracing Strategies* by Tarik Soulami
- *Advanced Windows Debugging* by Mario Hewardt and Daniel Pravat
- Defrag Tools, episodes 13 through 29, about WinDbg

See also

- Getting started with WinDbg (kernel-mode)
- Getting started with WinDbg (user-mode)
- Choosing the 32-Bit or 64-Bit debugging tools
- Debugging environments
- Setting up debugging (kernel-mode and user-mode)
- Debug universal drivers - step by step lab (echo kernel-mode)
- Debug drivers - step by step lab (Sysvad kernel-mode)

Getting Started with WinDbg (User-Mode)

7/29/2019 • 6 minutes to read • Edit Online

WinDbg is a kernel-mode and user-mode debugger that is included in Debugging Tools for Windows. Here we provide hands-on exercises that will help you get started using WinDbg as a user-mode debugger.

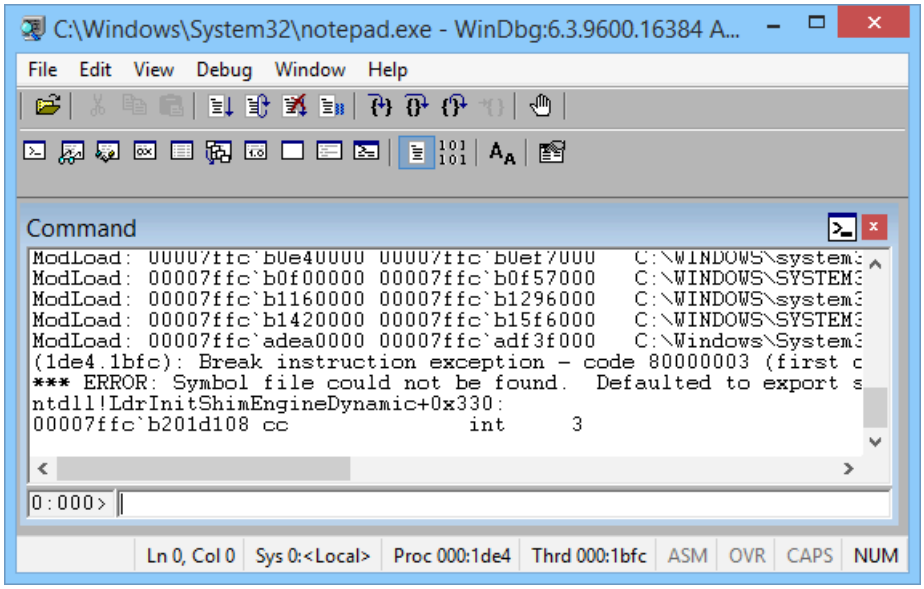
For information about how to get Debugging Tools for Windows, see Debugging Tools for Windows (WinDbg, KD, CDB, NTSD).

After you have installed the debugging tools, locate the installation directories for 64-bit (x64) and 32-bit (x86) versions of the tools. For example:

- C:\Program Files (x86)\Windows Kits\8.1\Debuggers\x64
- C:\Program Files (x86)\Windows Kits\8.1\Debuggers\x86

Launch Notepad and attach WinDbg

1. Navigate to your installation directory, and open WinDbg.exe.
2. The debugger documentation is also available on line here.
3. On the **File** menu, choose **Open Executable**. In the Open Executable dialog box, navigate to the folder that contains notepad.exe (for example, C:\Windows\System32). For **File name**, enter notepad.exe. Click **Open**.



4. Near the bottom of the WinDbg window, in the command line, enter this command:

   .sympath srv*

   The output is similar to this:

```
Symbol search path is: srv*
Expanded Symbol search path is: cache*;SRV
```

   The symbol search path tells WinDbg where to look for symbol (PDB) files. The debugger needs symbol files to obtain information about code modules (function names, variable names, and the like).

   Enter this command, which tells WinDbg to do its initial finding and loading of symbol files:

.reload

5. To see the symbols for the Notepad.exe module, enter this command:

   x notepad!*

   Note If you don't see any output, enter **.reload** again.

   To see symbols in the Notepad.exe module that contain main, enter this command:

   x notepad!*main*

   The output is similar to this:

   ```
   000000d0`428ff7e8 00007ff6`3282122f notepad!WinMain
   ...
   ```

6. To put a breakpoint at notepad!WinMain, enter this command:

   bu notepad!WinMain

   To verify that your breakpoint was set, enter this command:

   bl

   The output is similar to this:

   ```
   0 e 00007ff6`32825f64     0001 (0001)  0:**** notepad!WinMain
   ```

7. To start Notepad running, enter this command:

   g

   Notepad runs until it comes to the **WinMain** function, and then breaks in to the debugger.

   ```
   Breakpoint 0 hit
   notepad!WinMain:
   00007ff6`32825f64 488bc4          mov     rax,rsp
   ```

   To see a list of code modules that are loaded in the Notepad process, enter this command:

   lm

   The output is similar to this:

```
0:000> lm
start             end                 module name
00007ff6`32820000 00007ff6`3285a000   notepad    (pdb symbols)          C:\...\notepad.pdb
00007ffc`ab7e0000 00007ffc`ab85b000   WINSPOOL   (deferred)
00007ffc`aba10000 00007ffc`abc6a000   COMCTL32   (deferred)
00007ffc`adea0000 00007ffc`adf3f000   SHCORE     (deferred)
00007ffc`af490000 00007ffc`af59f000   KERNELBASE   (deferred)
00007ffc`af7d0000 00007ffc`af877000   msvcrt     (deferred)
00007ffc`af880000 00007ffc`b0c96000   SHELL32    (deferred)
00007ffc`b0e40000 00007ffc`b0ef7000   OLEAUT32   (deferred)
00007ffc`b0f00000 00007ffc`b0f57000   sechost    (deferred)
00007ffc`b0f60000 00007ffc`b1005000   ADVAPI32   (deferred)
00007ffc`b1010000 00007ffc`b1155000   GDI32      (deferred)
00007ffc`b1160000 00007ffc`b1296000   RPCRT4     (deferred)
00007ffc`b12a0000 00007ffc`b1411000   USER32     (deferred)
00007ffc`b1420000 00007ffc`b15f6000   combase    (deferred)
00007ffc`b16c0000 00007ffc`b17f9000   MSCTF      (deferred)
00007ffc`b1800000 00007ffc`b189a000   COMDLG32   (deferred)
00007ffc`b18a0000 00007ffc`b18f1000   SHLWAPI    (deferred)
00007ffc`b1b60000 00007ffc`b1cd8000   ole32      (deferred)
00007ffc`b1cf0000 00007ffc`b1e2a000   KERNEL32   (pdb symbols)          C:\...\kernel32.pdb
00007ffc`b1eb0000 00007ffc`b1ee4000   IMM32      (deferred)
00007ffc`b1f50000 00007ffc`b20fa000   ntdll      (private pdb symbols)  C:\...\ntdll.pdb
```

To see a stack trace, enter this command:

k

The output is similar to this:

```
0:000> k
Child-SP          RetAddr           Call Site
00000048`4e0cf6a8 00007ff6`3282122f notepad!WinMain
00000048`4e0cf6b0 00007ffc`b1cf16ad notepad!WinMainCRTStartup+0x1a7
00000048`4e0cf770 00007ffc`b1fc4629 KERNEL32!BaseThreadInitThunk+0xd
00000048`4e0cf7a0 00000000`00000000 ntdll!RtlUserThreadStart+0x1d ...
```

8. To start Notepad running again, enter this command:

   g

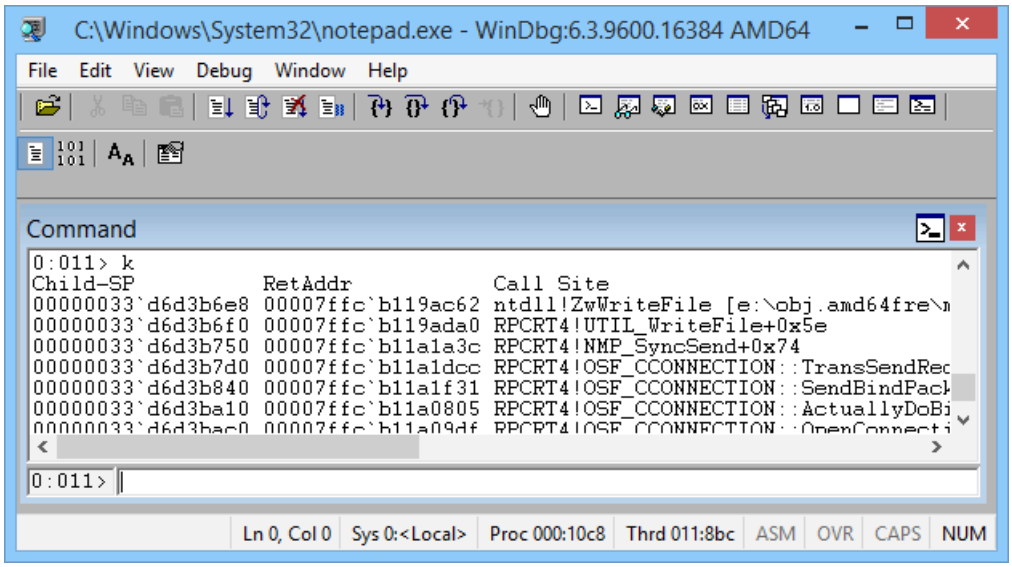
9. To break in to Notepad, choose **Break** from the **Debug** menu.

10. To set and verify a breakpoint at **ZwWriteFile**, enter these commands:

    bu ntdll!ZwWriteFile

    bl

11. Enter g to start Notepad running again. In the Notepad window, enter some text and choose **Save** from the **File** menu. The running code breaks in when it comes to **ZwCreateFile**. Enter k to see the stack trace.



In the WinDbg window, just to the left of the command line, notice the processor and thread numbers. In this example the current processor number is 0, and the current thread number is 11. So we are looking at the stack trace for thread 11 (which happens to be running on processor 0).

12. To see a list of all threads in the Notepad process, enter this command (the tilde):

    ~

    The output is similar to this:

```
0:011> ~
   0  Id: 10c8.128c Suspend: 1 Teb: 00007ff6`31cdd000 Unfrozen
   1  Id: 10c8.1a10 Suspend: 1 Teb: 00007ff6`31cdb000 Unfrozen
   2  Id: 10c8.1850 Suspend: 1 Teb: 00007ff6`31cd9000 Unfrozen
   3  Id: 10c8.1774 Suspend: 1 Teb: 00007ff6`31cd7000 Unfrozen
   4  Id: 10c8.1e80 Suspend: 1 Teb: 00007ff6`31cd5000 Unfrozen
   5  Id: 10c8.10ac Suspend: 1 Teb: 00007ff6`31cd3000 Unfrozen
   6  Id: 10c8.13a4 Suspend: 1 Teb: 00007ff6`31bae000 Unfrozen
   7  Id: 10c8.2b4 Suspend: 1 Teb: 00007ff6`31bac000 Unfrozen
   8  Id: 10c8.1df0 Suspend: 1 Teb: 00007ff6`31baa000 Unfrozen
   9  Id: 10c8.1664 Suspend: 1 Teb: 00007ff6`31ba8000 Unfrozen
  10  Id: 10c8.15e4 Suspend: 1 Teb: 00007ff6`31ba6000 Unfrozen
. 11  Id: 10c8.8bc Suspend: 1 Teb: 00007ff6`31ba4000 Unfrozen
```

    In this example, there are 12 threads with indexes 0 through 11.

13. To look at the stack trace for thread 0, enter these commands:

    ~0s

    k

    The output is similar to this:

```
0:011> ~0s
USER32!SystemParametersInfoW:
00007ffc`b12a4d20 48895c2408      mov     qword ptr [rsp+8], ...
0:000> k
Child-SP          RetAddr           Call Site
00000033`d1e9da48 00007ffc`adfb227d USER32!SystemParametersInfoW
(Inline Function) --------`-------- uxtheme!IsHighContrastMode+0x1d
00000033`d1e9da50 00007ffc`adfb2f12 uxtheme!IsThemeActive+0x4d
...
00000033`d1e9f810 00007ffc`b1cf16ad notepad!WinMainCRTStartup+0x1a7
00000033`d1e9f8d0 00007ffc`b1fc4629 KERNEL32!BaseThreadInitThunk+0xd
00000033`d1e9f900 00000000`00000000 ntdll!RtlUserThreadStart+0x1d
```

14. To quit debugging and detach from the Notepad process, enter this command:

    qd

Launch your own application and attach WinDbg

Suppose you have written and built this small console application.

```
...
void MyFunction(long p1, long p2, long p3)
{
    long x = p1 + p2 + p3;
    long y = 0;
    y = x / p2;
}

void main ()
{
    long a = 2;
    long b = 0;
    MyFunction(a, b, 5);
}
```

For this exercise, we will assume that the built application (MyApp.exe) and the symbol file (MyApp.pdb) are in C:\MyApp\x64\Debug. We will also assume that the application source code is in C:\MyApp\MyApp and that the target machine compiled MyApp.exe.

1. Open WinDbg.
2. On the **File** menu, choose **Open Executable**. In the Open Executable dialog box, navigate to C:\MyApp\x64\Debug. For **File name**, enter MyApp.exe. Click **Open**.
3. Enter these commands:

   .symfix

   .sympath+ C:\MyApp\x64\Debug

   Now WinDbg knows where to find symbols and source code for your application. In this case, the source code location doesn't need to be set with .srcpath because the symbols have fully qualified paths to the source files.

4. Enter these commands:

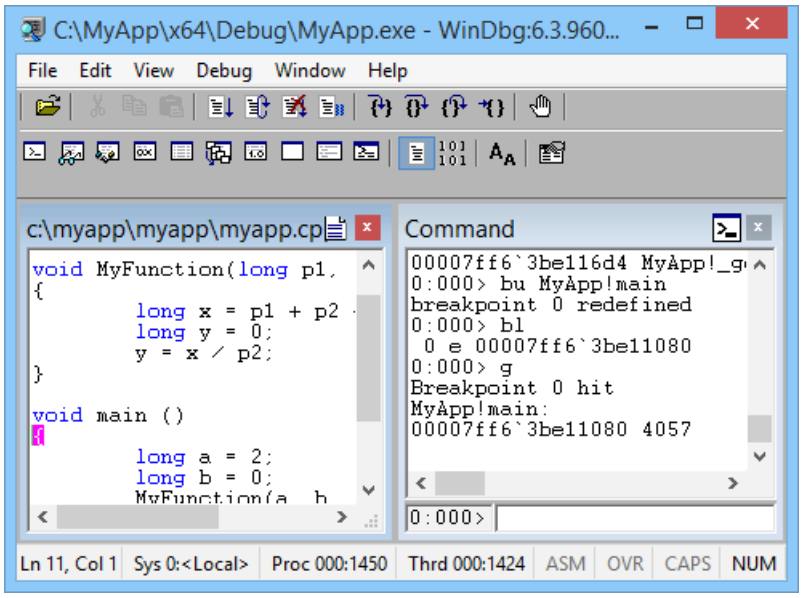
   .reload

   bu MyApp!main

g

Your application breaks in to the debugger when it comes to its **main** function.

WinDbg displays your source code and the Command window.



5. On the **Debug** menu, choose **Step Into** (or press **F11**). Continue stepping until you have stepped into **MyFunction**. When you step into the line `y = x / p2`, your application will crash and break in to the debugger. The output is similar to this:

```
(1450.1424): Integer divide-by-zero - code c0000094 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
MyApp!MyFunction+0x44:
00007ff6`3be11064 f77c2428    idiv  eax,dword ptr [rsp+28h] ss:00000063`2036f808=00000000
```

6. Enter this command:

!analyze -v

WinDbg displays an analysis of the problem (division by 0 in this case).

```
FAULTING_IP:
MyApp!MyFunction+44 [c:\myapp\myapp\myapp.cpp @ 7]
00007ff6`3be11064 f77c2428        idiv    eax,dword ptr [rsp+28h]

EXCEPTION_RECORD:  ffffffffffffffff -- (.exr 0xffffffffffffffff)
ExceptionAddress: 00007ff63be11064 (MyApp!MyFunction+0x0000000000000044)
   ExceptionCode: c0000094 (Integer divide-by-zero)
  ExceptionFlags: 00000000
NumberParameters: 0
...
STACK_TEXT:
00000063`2036f7e0 00007ff6`3be110b8 : ... : MyApp!MyFunction+0x44
00000063`2036f800 00007ff6`3be1141d : ... : MyApp!main+0x38
00000063`2036f840 00007ff6`3be1154e : ... : MyApp!__tmainCRTStartup+0x19d
00000063`2036f8b0 00007ffc`b1cf16ad : ... : MyApp!mainCRTStartup+0xe
00000063`2036f8e0 00007ffc`b1fc4629 : ... : KERNEL32!BaseThreadInitThunk+0xd
00000063`2036f910 00000000`00000000 : ... : ntdll!RtlUserThreadStart+0x1d

STACK_COMMAND: dt ntdll!LdrpLastDllInitializer BaseDllName ;dt ntdll!LdrpFailureData ;.cxr 0x0 ;kb

FOLLOWUP_IP:
MyApp!MyFunction+44 [c:\myapp\myapp\myapp.cpp @ 7]
00007ff6`3be11064 f77c2428        idiv    eax,dword ptr [rsp+28h]

FAULTING_SOURCE_LINE:  c:\myapp\myapp\myapp.cpp

FAULTING_SOURCE_FILE:  c:\myapp\myapp\myapp.cpp

FAULTING_SOURCE_LINE_NUMBER:  7

FAULTING_SOURCE_CODE:
     3: void MyFunction(long p1, long p2, long p3)
     4: {
     5:     long x = p1 + p2 + p3;
     6:     long y = 0;
>    7:  y = x / p2;
     8: }
     9:
    10: void main ()
    11: {
    12:     long a = 2;
...
```

Summary of commands

- **Contents** command on the **Help** menu
- .sympath (Set Symbol Path)
- .reload (Reload Module)
- x (Examine Symbols)
- g (Go)
- **Break** command on the **Debug** menu
- lm (List Loaded Modules)
- k (Display Stack Backtrace)
- bu (Set Breakpoint)
- bl (Breakpoint List)
- ~ (Thread Status)
- ~s (Set Current Thread)
- .sympath+ (Set Symbol Path) append to existing symbol path
- .srcpath (Set Source Path)

- **Step Into** command on the **Debug** menu (**F11**)
- !analyze -v
- qd (Quit and Detach)

Related topics

Getting Started with WinDbg (Kernel-Mode)

Debugger Operation

Debugging Techniques

Debugging Tools for Windows (WinDbg, KD, CDB, NTSD)

Getting Started with WinDbg (Kernel-Mode)

12/5/2018 • 8 minutes to read • Edit Online

WinDbg is a kernel-mode and user-mode debugger that is included in Debugging Tools for Windows. Here we provide hands-on exercises that will help you get started using WinDbg as a kernel-mode debugger.

For information about how to get Debugging Tools for Windows, see Debugging Tools for Windows (WinDbg, KD, CDB, NTSD). After you have installed the debugging tools, locate the installation directories for 64-bit (x64) and 32-bit (x86) versions of the tools. For example:

- C:\Program Files (x86)\Windows Kits\8.1\Debuggers\x64
- C:\Program Files (x86)\Windows Kits\8.1\Debuggers\x86

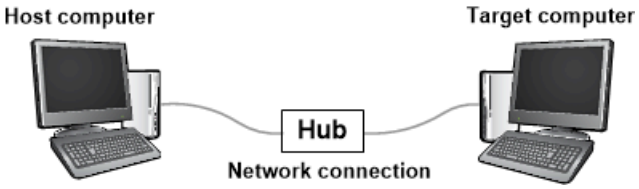
Set up a kernel-mode debugging

A kernel-mode debugging environment typically has two computers: the *host computer* and the *target computer*. The debugger runs on the host computer, and the code being debugged runs on the target computer. The host and target are connected by a debug cable.

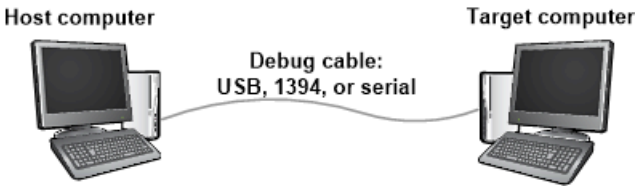
The Windows debuggers support these types of cables for debugging:

- Ethernet
- USB 2.0
- USB 3.0
- 1394
- Serial (also called null modem)

If your target computer is running Windows 8 or later, you can use any type of debug cable, including Ethernet. This diagram illustrates a host and target computer connected for debugging over Ethernet cable.



If your target computer is running a version of Windows earlier than Window 8, then you cannot use Ethernet for debugging; you must use USB, 1394, or serial. This diagram illustrates a host and target computer connected by a USB, 1394, or serial debug cable.



For details about how to set up the host and target computers, see Setting Up Kernel-Mode Debugging Manually.

Establish a kernel-mode debugging session

After you have set up your host and target computer and connected them with a debug cable, you can establish a kernel-mode debugging session by following the instructions in the same topic that you used for getting set up.

For example, if you decided to set up your host and target computers for debugging over Ethernet, you can find instructions for establishing a kernel-mode debugging session is this topic:

- Setting Up KDNET Network Kernel Debugging Automatically

Get started using WinDbg

1. On the host computer, open WinDbg and establish a kernel-mode debugging session with the target computer.
2. In WinDbg, choose **Contents** from the **Help** menu. This opens the debugger documentation CHM file. The debugger documentation is also available on line here.
3. When you establish a kernel-mode debugging session, WinDbg might break in to the target computer automatically. If WinDbg has not already broken in, choose **Break** from the **Debug** menu.
4. Near the bottom of the WinDbg window, in the command line, enter this command:

   .sympath srv*

   The output is similar to this:

   ```
   Symbol search path is: srv*
   Expanded Symbol search path is: cache*;SRV*https://msdl.microsoft.com/download/symbols
   ```

   The symbol search path tells WinDbg where to look for symbol (PDB) files. The debugger needs symbol files to obtain information about code modules (function names, variable names, and the like).

   Enter this command, which tells WinDbg to do its initial finding and loading of symbol files:

   .reload

5. To see a list of loaded modules, enter this command:

   lm

   The output is similar to this:

   ```
   0:000>3: kd> lm
   start             end                 module name
   fffff800`00000000 fffff800`00088000   CI         (deferred)
   ...
   fffff800`01143000 fffff800`01151000   BasicRender   (deferred)
   fffff800`01151000 fffff800`01163000   BasicDisplay  (deferred)
   ...
   fffff800`02a0e000 fffff800`03191000   nt  (pdb symbols) C:\...\ntkrnlmp.pdb
   fffff800`03191000 fffff800`03200000   hal (deferred)
   ...
   ```

6. To start target computer running, enter this command:

   g

7. To break in again, choose **Break** from the **Debug** menu.
8. Enter this command to examine the _FILE_OBJECT data type in the nt module:

   dt nt!_FILE_OBJECT

   The output is similar to this:

```
0:000>0: kd> dt nt!_FILE_OBJECT
   +0x000 Type             : Int2B
   +0x002 Size             : Int2B
   +0x008 DeviceObject     : Ptr64 _DEVICE_OBJECT
   +0x010 Vpb              : Ptr64 _VPB
   ...
   +0x0c0 IrpList          : _LIST_ENTRY
   +0x0d0 FileObjectExtension : Ptr64 Void
```

9. Enter this command to examine some of the symbols in the nt module:

   x nt!*CreateProcess*

   The output is similar to this:

```
0:000>0: kd> x nt!*CreateProcess*
fffff800`030821cc nt!ViCreateProcessCallbackInternal (<no parameter info>)
...
fffff800`02e03904 nt!MmCreateProcessAddressSpace (<no parameter info>)
fffff800`02cece00 nt!PspCreateProcessNotifyRoutine = <no type information>
...
```

10. Enter this command to put a breakpoint at **MmCreateProcessAddressSpace**:

    bu nt!MmCreateProcessAddressSpace

    To verify that the breakpoint is set, enter this command:

    bl

    The output is similar to this:

```
0:000>0: kd> bu nt!MmCreateProcessAddressSpace
0: kd> bl
0 e fffff800`02e03904     0001 (0001) nt!MmCreateProcessAddressSpace
```

    Enter **g** to let the target computer run.

11. If the target computer doesn't break in to the debugger immediately, perform a few actions on the target computer (for example, open Notepad). The target computer will break in to the debugger when **MmCreateProcessAddressSpace** is called. To see the stack trace, enter these commands:

    .reload

    k

    The output is similar to this:

```
0:000>2: kd> k
Child-SP          RetAddr           Call Site
ffffd000`224b4c88 fffff800`02d96834 nt!MmCreateProcessAddressSpace
ffffd000`224b4c90 fffff800`02dfef17 nt!PspAllocateProcess+0x5d4
ffffd000`224b5060 fffff800`02b698b3 nt!NtCreateUserProcess+0x55b
...
000000d7`4167fbb0 00007ffd`14b064ad KERNEL32!BaseThreadInitThunk+0xd
000000d7`4167fbe0 00000000`00000000 ntdll!RtlUserThreadStart+0x1d
```

12. On the **View** menu, choose **Disassembly**.

    On the **Debug** menu, choose **Step Over** (or press **F10**). Enter step commands a few more times as you

watch the Disassembly window.

13. Clear your breakpoint by entering this command:

    bc *

    Enter **g** to let the target computer run. Break in again by choosing **Break** from the **Debug** menu or pressing **CTRL-Break**.

14. To see a list of all processes, enter this command:

    !process 0 0

    The output is similar to this:

```
0:000>0: kd> !process 0 0
**** NT ACTIVE PROCESS DUMP ****
PROCESS ffffe000002287c0
    SessionId: none  Cid: 0004    Peb: 00000000  ParentCid: 0000
    DirBase: 001aa000  ObjectTable: ffffc00000003000  HandleCount: <Data Not Accessible>
    Image: System

PROCESS ffffe00001e5a900
    SessionId: none  Cid: 0124    Peb: 7ff7809df000  ParentCid: 0004
    DirBase: 100595000  ObjectTable: ffffc000002c5680  HandleCount: <Data Not Accessible>
    Image: smss.exe
...
PROCESS ffffe00000d52900
    SessionId: 1  Cid: 0910    Peb: 7ff669b8e000  ParentCid: 0a98
    DirBase: 3fdba000  ObjectTable: ffffc00007bfd540  HandleCount: <Data Not Accessible>
    Image: explorer.exe
```

15. Copy the address of one process, and enter this command:

    !process *Address* **2**

    For example: **!process ffffe00000d5290 2**

    The output shows the threads in the process.

```
0:000>0:000>0: kd> !process ffffe00000d52900 2
PROCESS ffffe00000d52900
    SessionId: 1  Cid: 0910    Peb: 7ff669b8e000  ParentCid: 0a98
    DirBase: 3fdba000  ObjectTable: ffffc00007bfd540  HandleCount:
    Image: explorer.exe

        THREAD ffffe00000a0d880  Cid 0910.090c  Teb: 00007ff669b8c000
            ffffe00000d57700  SynchronizationEvent

        THREAD ffffe00000e48880  Cid 0910.0ad8  Teb: 00007ff669b8a000
            ffffe00000d8e230  NotificationEvent
            ffffe00000cf6870  Semaphore Limit 0xffff
            ffffe000039c48c0  SynchronizationEvent
        ...
        THREAD ffffe00000e6d080  Cid 0910.0cc0  Teb: 00007ff669a10000
            ffffe0000089a300  QueueObject
```

16. Copy the address of one thread, and enter this command:

    !thread *Address*

    For example: **!thread ffffe00000e6d080**

    The output shows information about the individual thread.

```
0: kd> !thread ffffe00000e6d080
THREAD ffffe00000e6d080  Cid 0910.0cc0  Teb: 00007ff669a10000 Win32Thread: 0000000000000000 WAIT: ...
    ffffe0000089a300  QueueObject
Not impersonating
DeviceMap                 ffffc000034e7840
Owning Process            ffffe00000d52900       Image:         explorer.exe
Attached Process          N/A            Image:         N/A
Wait Start TickCount      13777          Ticks: 2 (0:00:00:00.031)
Context Switch Count      2              IdealProcessor: 1
UserTime                  00:00:00.000
KernelTime                00:00:00.000
Win32 Start Address ntdll!TppWorkerThread (0x00007ffd14ab2850)
Stack Init ffffd00021bf1dd0 Current ffffd00021bf1580
Base ffffd00021bf2000 Limit ffffd00021bec000 Call 0
Priority 13 BasePriority 13 UnusualBoost 0 ForegroundBoost 0 IoPriority 2 PagePriority 5
...
```

17. To see all the device nodes in the Plug and Play device tree, enter this command:

    !devnode 0 1

```
0:000>0: kd> !devnode 0 1
Dumping IopRootDeviceNode (= 0xffffe000002dbd30)
DevNode 0xffffe000002dbd30 for PDO 0xffffe000002dc9e0
  InstancePath is "HTREE\ROOT\0"
  State = DeviceNodeStarted (0x308)
  Previous State = DeviceNodeEnumerateCompletion (0x30d)
  DevNode 0xffffe000002d9d30 for PDO 0xffffe000002daa40
    InstancePath is "ROOT\volmgr\0000"
    ServiceName is "volmgr"
    State = DeviceNodeStarted (0x308)
    Previous State = DeviceNodeEnumerateCompletion (0x30d)
    DevNode 0xffffe00001d49290 for PDO 0xffffe000002a9a90
      InstancePath is "STORAGE\Volume\{3007dfd3-df8d-11e3-824c-806e6f6e6963}#0000000000100000"
      ServiceName is "volsnap"
      TargetDeviceNotify List - f 0xffffc0000031b520  b 0xffffc0000008d0f0
      State = DeviceNodeStarted (0x308)
      Previous State = DeviceNodeStartPostWork (0x307)
...
```

18. To see the device nodes along with their hardware resources, enter this command:

    !devnode 0 9

```
0:000>...
        DevNode 0xffffe000010fa770 for PDO 0xffffe000010c2060
          InstancePath is "PCI\VEN_8086&DEV_2937&SUBSYS_2819103C&REV_02\3&33fd14ca&0&D0"
          ServiceName is "usbuhci"
          State = DeviceNodeStarted (0x308)
          Previous State = DeviceNodeEnumerateCompletion (0x30d)
          TranslatedResourceList at 0xffffc00003c78b00  Version 1.1  Interface 0x5  Bus #0
            Entry 0 - Port (0x1) Device Exclusive (0x1)
              Flags (0x131) - PORT_MEMORY PORT_IO 16_BIT_DECODE POSITIVE_DECODE
              Range starts at 0x3120 for 0x20 bytes
            Entry 1 - DevicePrivate (0x81) Device Exclusive (0x1)
              Flags (0000) -
              Data - {0x00000001, 0x00000004, 0000000000}
            Entry 2 - Interrupt (0x2) Shared (0x3)
              Flags (0000) - LEVEL_SENSITIVE
              Level 0x8, Vector 0x81, Group 0, Affinity 0xf
...
```

19. To see a device node that has a service name of disk, enter this command:

!devnode 0 1 disk

```
0: kd> !devnode 0 1 disk
Dumping IopRootDeviceNode (= 0xffffe000002dbd30)
DevNode 0xffffe0000114fd30 for PDO 0xffffe00001159610
  InstancePath is "IDE\DiskST3250820AS_____________________________3.CHL___\5&14544e82&0&0.0.0"
  ServiceName is "disk"
  State = DeviceNodeStarted (0x308)
  Previous State = DeviceNodeEnumerateCompletion (0x30d)
...
```

20. The output of **!devnode 0 1** displays the address of the physical device object (PDO) for the node. Copy the address of a physical device object (PDO), and enter this command:

    !devstack *PdoAddress*

    For example: *PdoAddress***!devstack 0xffffe00001159610**

```
0:000>0: kd> !devstack 0xffffe00001159610
  !DevObj           !DrvObj            !DevExt           ObjectName
  ffffe00001d50040  \Driver\partmgr    ffffe00001d50190
  ffffe00001d51450  \Driver\disk       ffffe00001d515a0  DR0
  ffffe00001156e50  \Driver\ACPI       ffffe000010d8bf0
```

21. To get information about the driver disk.sys, enter this command:

    !drvobj disk 2

```
0:000>0: kd> !drvobj disk 2
Driver object (ffffe00001d52680) is for:
 \Driver\disk
DriverEntry:   fffff800006b1270 disk!GsDriverEntry
DriverStartIo: 00000000
DriverUnload:  fffff800010b0b5c CLASSPNP!ClassUnload
AddDevice:     fffff800010aa110 CLASSPNP!ClassAddDevice

Dispatch routines:
[00] IRP_MJ_CREATE                      fffff8000106d160    CLASSPNP!ClassGlobalDispatch
[01] IRP_MJ_CREATE_NAMED_PIPE           fffff80002b0ab24    nt!IopInvalidDeviceRequest
[02] IRP_MJ_CLOSE                       fffff8000106d160    CLASSPNP!ClassGlobalDispatch
[03] IRP_MJ_READ                        fffff8000106d160    CLASSPNP!ClassGlobalDispatch
...
[1b] IRP_MJ_PNP                         fffff8000106d160    CLASSPNP!ClassGlobalDispatch
```

22. The output of !drvobj displays addresses of dispatch routines: for example, CLASSPNP!ClassGlobalDispatch. To set and verify a breakpoint at ClassGlobalDispatch, enter these commands:

    bu CLASSPNP!ClassGlobalDispatch

    bl

    Enter g to let the target computer run.

    If the target computer doesn't break in to the debugger immediately, perform a few actions on the target computer (for example, open Notepad and save a file). The target computer will break in to the debugger when **ClassGlobalDispatch** is called. To see the stack trace, enter these commands:

    .reload

    k

The output is similar to this:

```
2: kd> k
Child-SP          RetAddr           Call Site
ffffd000`21d06cf8 fffff800`0056c14e CLASSPNP!ClassGlobalDispatch
ffffd000`21d06d00 fffff800`00f2c31d volmgr!VmReadWrite+0x13e
ffffd000`21d06d40 fffff800`0064515d fvevol!FveFilterRundownReadWrite+0x28d
ffffd000`21d06e20 fffff800`0064578b rdyboost!SmdProcessReadWrite+0x14d
ffffd000`21d06ef0 fffff800`00fb06ad rdyboost!SmdDispatchReadWrite+0x8b
ffffd000`21d06f20 fffff800`0085cef5 volsnap!VolSnapReadFilter+0x5d
ffffd000`21d06f50 fffff800`02b619f7 Ntfs!NtfsStorageDriverCallout+0x16
...
```

23. To end your debugging session, enter this command:

    qd

Summary of commands

- **Contents** command on the **Help** menu
- .sympath (Set Symbol Path)
- .reload (Reload Module)
- x (Examine Symbols)
- g (Go)
- dt (Display Type)
- **Break** command on the **Debug** menu
- lm (List Loaded Modules)
- k (Display Stack Backtrace)
- bu (Set Breakpoint)
- bl (Breakpoint List)
- bc (Breakpoint Clear)
- **Step Into** command on the **Debug** menu (**F11**)
- !process
- !thread
- !devnode
- !devstack
- !drvobj
- qd (Quit and Detach)

Related topics

Getting Started with WinDbg (User-Mode)

Setting Up Kernel-Mode Debugging Manually

Debugger Operation

Debugging Techniques

Debugging Tools for Windows (WinDbg, KD, CDB, NTSD)

Choosing the 32-Bit or 64-Bit Debugging Tools

12/5/2018 • 2 minutes to read • Edit Online

When you install Debugging Tools for Windows, you get both a 32-bit set of tools and a 64-bit set of tools. If you use the Microsoft Visual Studio debugging environment, you don't have to be concerned about whether to use the 32- or 64-bit set because Visual Studio automatically chooses the correct debugging tools.

If you are using one of the other debugging environments (WinDbg, KD, CDB, or NTSD), you have to make the choice yourself. To determine which set of debugging tools to use, you need to know the type of processor that is running on your host computer and whether the host computer is running a 32- or 64-bit version of Windows.

The computer that runs the debugger is called the *host computer*, and the computer being debugged is called the *target computer*.

Host computer running a 32-bit version of Windows

If your host computer is running a 32-bit version of Windows, use the 32-bit debugging tools. (This situation applies to both x86-based and x64-based targets.)

x64-based host computer running a 64-bit version of Windows

If your host computer uses an x64-based processor and is running a 64-bit version of Windows, the following rules apply:

- If you are analyzing a dump file, you can use either the 32-bit debugging tools or the 64-bit debugging tools. (It is not important whether the dump file is a user-mode dump file or a kernel-mode dump file, and it is not important whether the dump file was made on an x86-based or an x64-based platform.)
- If you are performing live kernel-mode debugging, you can use either the 32-bit debugging tools or the x64 debugging tools. (This situation applies to both x86-based and x64-based targets.)
- If you are debugging live user-mode code that is running on the same computer as the debugger, use the 64-bit tools for debugging 64-bit code and 32-bit code running on WOW64. To set the debugger for 32-bit or 64-bit mode, use the **.effmach** command.
- If you are debugging live 32-bit user-mode code that is running on a separate target computer, use the 32-bit debugging tools.

Related topics

Windows Debugging

Debugging Environments

12/5/2018 • 5 minutes to read • Edit Online

Starting with Windows Driver Kit (WDK) 8.0, the driver development environment and the Windows debugger are integrated into Microsoft Visual Studio.

After you install Visual Studio and the WDK, you have six available debugging environments:

- Visual Studio with integrated Windows debugger
- Microsoft Windows Debugger (WinDbg)
- Microsoft Kernel Debugger (KD)
- NTKD
- Microsoft Console Debugger (CDB)
- Microsoft NT Symbolic Debugger (NTSD)

The following sections describe the debugging environments.

Visual Studio with integrated Windows debugger

Starting with WDK 8.0, the driver development environment and the Windows debugger are integrated into Visual Studio. In this integrated environment, most of the tools you need for coding, building, packaging, testing, debugging, and deploying a driver are available in the Visual Studio user interface.

Typically kernel-mode debugging requires two computers. The debugger runs on the *host computer* and the code being debugged runs on the *target computer*. With the Windows debugger integrated into Visual Studio, you can perform a wide variety of debugging tasks, including those shown in the following list, from the host computer.

- Configure a set of target computers for debugging.
- Configure the debugging connections to a set of target computers.
- Launch a kernel-mode debugging session between the host computer and a target computer.
- Debug a user-mode process on the host computer.
- Debug a user-mode process on a target computer.
- Connect to remote debugging sessions.
- View assembly code and source code.
- View and manipulate local variables, parameters, and other symbols.
- View and manipulate memory.
- Navigate call stacks.
- Set breakpoints.
- Execute debugger commands.

You can also build drivers, deploy drivers, and run driver tests from within the Visual Studio user interface. If you make use of the Visual Studio integration provided by both the WDK and Debugging Tools for Windows, you can perform almost all of the driver development, packaging, deployment, testing, and debugging tasks from within Visual Studio on the host computer. Here are some of the WDK capabilities that have been integrated into Visual Studio.

- Configure a set of target computers for driver testing.
- Create and sign a driver package.
- Deploy a driver package to a target computer.
- Install and load a driver on a target computer.

- Test a driver on a target computer.

WinDbg

Microsoft Windows Debugger (WinDbg) is a powerful Windows-based debugger that is capable of both user-mode and kernel-mode debugging. WinDbg provides debugging for the Windows kernel, kernel-mode drivers, and system services, as well as user-mode applications and drivers.

WinDbg uses the Visual Studio debug symbol formats for source-level debugging. It can access any symbol or variable from a module that has PDB symbol files, and can access any public function's name that is exposed by modules that were compiled with COFF symbol files (such as Windows .dbg files).

WinDbg can view source code, set breakpoints, view variables (including C++ objects), stack traces, and memory. Its Debugger Command window allows the user to issue a wide variety of commands.

For kernel-mode debugging, WinDbg typically requires two computers (the host computer and the target computer). WinDbg also supports various remote debugging options for both user-mode and kernel-mode targets.

WinDbg is a graphical-interface counterpart to CDB/NTSD and to KD/NTKD.

KD

Microsoft Kernel Debugger (KD) is a character-based console program that enables in-depth analysis of kernel-mode activity on all NT-based operating systems. You can use KD can to debug kernel-mode components and drivers, or to monitor the behavior of the operating system itself. KD also supports multiprocessor debugging.

Typically, KD does not run on the computer being debugged. You need two computers (the *host computer* and the *target computer*) for kernel-mode debugging.

NTKD

There is a variation of the KD debugger named NTKD. It is identical to KD in every way, except that it spawns a new text window when it is started, whereas KD inherits the Command Prompt window from which it was invoked.

CDB

Microsoft Console Debugger (CDB) is a character-based console program that enables low-level analysis of Windows user-mode memory and constructs. The name *Console Debugger* is used to indicate the fact that CDB is classified as a console application; it does not imply that the target application must be a console application. In fact, CDB is fully capable of debugging both console applications and graphical Windows programs.

CDB is extremely powerful for debugging a program that is currently running or has recently crashed (live analysis), yet simple to set up. It can be used to investigate the behavior of a working application. In the case of a failing application, CDB can be used to obtain a stack trace or to look at the guilty parameters. It works well across a network (using a remote access server), as it is character-based.

With CDB, you can display and execute program code, set breakpoints, and examine and change values in memory. CDB can analyze binary code by disassembling it and displaying assembly instructions. It can also analyze source code directly.

Because CDB can access memory locations through addresses or global symbols, you can refer to data and instructions by name rather than by address, making it easy to locate and debug specific sections of code. CDB supports debugging multiple threads and processes. It is extensible, and can read and write both paged and non-paged memory.

If the target application is itself a console application, the target will share the console window with CDB. To spawn a separate console window for a target console application, use the *-2* command-line option.

NTSD

There is a variation of the CDB debugger named Microsoft NT Symbolic Debugger (NTSD). It is identical to CDB in every way, except that it spawns a new text window when it is started, whereas CDB inherits the Command Prompt window from which it was invoked.

Since the **start** command can also be used to spawn a new console window, the following two constructions will give the same results:

```
start cdb parameters
ntsd parameters
```

It is possible to redirect the input and output from NTSD (or CDB) so that it can be controlled from a kernel debugger (either Visual Studio, WinDbg, or KD). If this technique is used with NTSD, no console window will appear at all. Controlling NTSD from the kernel debugger is therefore especially useful, since it results in an extremely light-weight debugger that places almost no burden on the computer containing the target application. This combination can be used to debug system processes, shutdown, and the later stages of boot up. See Controlling the User-Mode Debugger from the Kernel Debugger for details.

Related topics

Windows Debugging

Setting Up Debugging (Kernel-Mode and User-Mode)

12/5/2018 • 2 minutes to read • Edit Online

There are two ways you can set up debugging with the Windows debuggers. You can use Microsoft Visual Studio (integrated with the Windows Driver Kit (WDK)), or you can do the setup manually. After you set up kernel-mode debugging, you can use Visual Studio, WinDbg, or KD to establish a debugging session. After you set up user-mode debugging, you can use Visual Studio, WinDbg, CDB, or NTSD to establish a debugging session.

Note The Windows debuggers are included in Debugging Tools for Windows. These debuggers are different from the Visual Studio debugger, which is included with Visual Studio. For more information, see Windows Debugging.

In this section

- Setting Up Kernel-Mode Debugging Manually
- Supported Ethernet NICs for Network Kernel Debugging in Windows 10
- Supported Ethernet NICs for Network Kernel Debugging in Windows 8.1
- Supported Ethernet NICs for Network Kernel Debugging in Windows 8
- Setting Up User-Mode Debugging in Visual Studio
- Setting Up Network Debugging of a Virtual Machine Host
- Configuring tools.ini
- Using KDbgCtrl

Setting Up Kernel-Mode Debugging

12/5/2018 • 2 minutes to read • Edit Online

This section describes how to set up kernel-mode debugging.

> **TIP**
> The recommended approach is to use network (KDNET) debugging and use the kdnet utility to configure that automatically. For more information, see Setting Up KDNET Network Kernel Debugging Automatically.

In this section

- Setting Up KDNET Network Kernel Debugging Automatically
- Setting Up KDNET Network Kernel Debugging Manually
- Setting Up Kernel-Mode Debugging over a USB 3.0 Cable Manually
- Setting Up Kernel-Mode Debugging over a USB 2.0 Cable Manually
- Setting Up Kernel-Mode Debugging over a Serial Cable Manually
- Setting Up Kernel-Mode Debugging using Serial over USB Manually
- Setting Up Kernel-Mode Debugging over a 1394 Cable Manually
- Setting Up Kernel-Mode Debugging of a Virtual Machine Host- KDNET
- Setting Up Kernel-Mode Debugging of a Virtual Machine Manually using a COM Port
- Setting Up Local Kernel Debugging of a Single Computer Manually

Related topics

Setting Up Debugging (Kernel-Mode and User-Mode)

Setting Up KDNET Network Kernel Debugging Automatically

9/20/2019 • 5 minutes to read • Edit Online

Debugging Tools for Windows supports kernel debugging over a network. This topic describes how to set up network debugging automatically using the kdnet.exe setup tool.

The computer that runs the debugger is called the *host computer*, and the computer being debugged is called the *target computer*. The host computer must be running Windows 7 or later, and the target computer must be running Windows 8 or later.

Determining the IP Address of the Host Computer

1. Confirm that the target and host PCs are connected to a network hub or switch using an appropriate network cable.
2. On the host computer, open a Command Prompt window and enter *IPConfig* to display the IP configuration.
3. In the command output, locate the IPv4 address of the Ethernet adapter.

```
...

Ethernet adapter Ethernet:
...

IPv4 Address. . . . . . . . . . . : <YourHostIPAddress>
...
```

4. Make a note of the IPv4 address of the network adapter that you intend to use for debugging.

Setting Up the Host and Target Computers

Use the kdnet.exe utility to automatically configure the debugger settings on the target PC, by following these steps.

1. Confirm that the Windows Debugging Tools are installed on the host system. For information on downloading and installing the debugger tools, see Download Debugging Tools for Windows.
2. Locate the *kdnet.exe* and *VerifiedNICList.xml* files. By default, they are located here.

```
C:\Program Files (x86)\Windows Kits\10\Debuggers\x64
```

> **NOTE**
>
> These directions assumes that both PCs are running a 64 bit version of Windows on both the target and host. If that is not the case, the best approach is to run the same "bitness" of tools on the host that the target is running. For example, if the target is running 32-bit Windows, run a 32 version of the debugger on the host. For more information, see Choosing the 32-Bit or 64-Bit Debugging Tools.

3. On the host computer, copy the two files to a network share or thumb drive, so that they will be available on the target computer.

4. On the target computer, create a C:\KDNET directory and copy the *kdnet.exe* and *VerifiedNICList.xml* files to that directory.

   > **IMPORTANT**
   > Before using kdnet to change boot information you may need to temporarily suspend Windows security features such as BitLocker and Secure Boot on the test PC. Re-enable these security features when testing is complete and appropriately manage the test PC, when the security features are disabled.

5. On the target computer, open a Command Prompt window as Administrator. Enter this command to verify that the target computer has a supported network adapter.

   ```
   C:\KDNET>kdnet
   Network debugging is supported on the following NICs:
   busparams=1.0.0, Broadcom NetXtreme Gigabit Ethernet, Plugged in.
   This Microsoft hypervisor supports using KDNET in guest VMs.
   ```

6. As the output from kdnet indicates that network adapter on the target is supported, we can proceed.

7. Type this command to set the IP address of the host system and generated a unique connection key. Use the IP address or the name of the host system. Pick a unique port address for each target/host pair that you work with, with in the recommended range of 50000-50039.

   ```
   C:\>kdnet <HostComputerIPAddress> <YourDebugPort>

   Enabling network debugging on Intel(R) 82577LM Gigabit Network Connection.
   Key=2steg4fzbj2sz.23418vzkd4ko3.1g34ou07z4pev.1sp3yo9yz874p
   ```

8. Copy the returned key into a notepad .txt file.

Using WinDbg

On the host computer, open WinDbg. On the **File** menu, choose **Kernel Debug**. In the Kernel Debugging dialog box, open the **Net** tab. Paste in your port number and key that you saved to in the notepad .txt file earlier. Click **OK**.

You can also start a WinDbg session by opening a Command Prompt window and entering the following command, where is the port you selected above, and is the key that was returned by kdnet above. Paste in the key in that you saved to in the notepad .txt file earlier.

```
windbg -k net:port=<YourDebugPort>,key=<YourKey>
```

If you are prompted about allowing WinDbg to access the port through the firewall, allow WinDbg to access the port for **all three** of the different network types.

At this point the debugger will be waiting for the target to reconnect and text similar to the following will be displayed in the debugger command window.

```
Microsoft (R) Windows Debugger Version 1.0.1908.30002 AMD64
Copyright (c) Microsoft Corporation. All rights reserved.

Using NET for debugging
Opened WinSock 2.0
Waiting to reconnect...
```

Restarting the Target PC

Once the debugger is connected, reboot the target computer. One way to do restart the PC, is to use this command from an administrator's command prompt.

```
shutdown -r -t 0
```

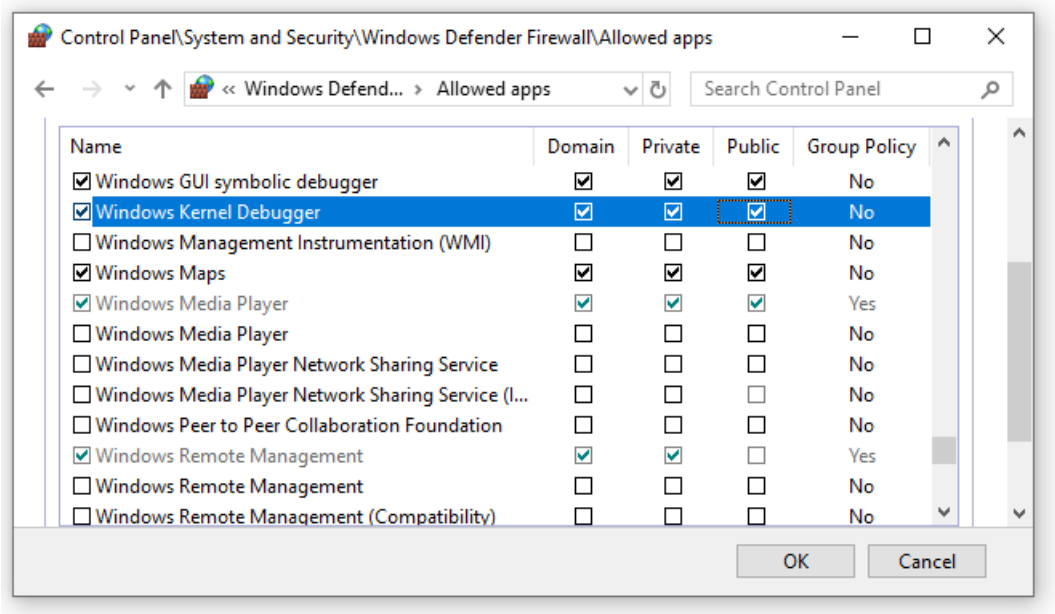
After the target PC restarts, the debugger should connect automatically.

Troubleshooting Tips

Debugging application must be allowed through firewall

The debugger must have access through the firewall. Use Control Panel to allow access through the firewall.

1. Open **Control Panel > System and Security** and click **Allow an app through Windows Firewall**.
2. In the list of applications, locate *Windows GUI Symbolic Debugger* and *Windows Kernel Debugger*.
3. Use the check boxes to allow those two applications **all three** of the different network types through the firewall.
4. Scroll down and click **OK**, to save the firewall changes. Restart the debugger.



Use Ping to test connectivity

If the debugger times out and does not connect, use the ping command on the target PC to verify connectivity.

```
C:\>Ping <HostComputerIPAddress>
```

Choosing a Port for Network Debugging

If the debugger times out and does not connect, it could be because the default port number of 50000 is already in use or it is blocked.

You can choose any port number from 49152 through 65535. The recommended range is between 50000 and 50039. The port that you choose will be opened for exclusive access by the debugger running on the host computer.

Note The range of port numbers that can be used for network debugging might be limited by your company's network policy. To determine whether your company's policy limits the range of ports that can be used for network debugging, check with your network administrators.

Supported Network Adapters

If "Network debugging is not supported on any of the NICs in this machine" is displayed when you run kdnet.exe, it means that the network adapter is not supported.

The host computer can use any network adapter, but the target computer must use a network adapter that is supported by Debugging Tools for Windows. For a list of supported network adapters, see Supported Ethernet NICs for Network Kernel Debugging in Windows 10 and Supported Ethernet NICs for Network Kernel Debugging in Windows 8.1.

Related topics

Supported Ethernet NICs for Network Kernel Debugging in Windows 10

Supported Ethernet NICs for Network Kernel Debugging in Windows 8.1

Setting Up KDNET Network Kernel Debugging Manually

Getting Started with WinDbg (Kernel-Mode)

Debug Universal Drivers - Step by Step Lab (Echo Kernel-Mode)

Setting Up KDNET Network Kernel Debugging Manually

12/7/2018 • 16 minutes to read • Edit Online

Debugging Tools for Windows supports kernel debugging over a network. This topic describes how to set up network debugging manually.

> **IMPORTANT**
>
> Setting up a network debugging manually is a complex and error prone process. To set up network debugging automatically, see Setting Up KDNET Network Kernel Debugging Automatically. Using the KDNET utility is **strongly** recommended for all debugger users.

The computer that runs the debugger is called the *host computer*, and the computer being debugged is called the *target computer*. The host computer must be running Windows 7 or later, and the target computer must be running Windows 8 or later.

Debugging over a network has the following advantages compared to debugging over other types of connectivity.

- The host and target computers can be anywhere on the local network.
- It is easy to debug many target computers from one host computer.
- Given any two computers, it is likely that they will both have Ethernet adapters. It is less likely that they will both have serial ports or both have 1394 ports.
- Network debugging is significantly faster than serial port debugging.

Supported Network Adapters

The host computer can use any network adapter, but the target computer must use a network adapter that is supported by Debugging Tools for Windows. For a list of supported network adapters, see Supported Ethernet NICs for Network Kernel Debugging in Windows 10 and Supported Ethernet NICs for Network Kernel Debugging in Windows 8.1.

Install the Debugging Tools for Windows

Confirm that the Debugging Tools for Windows are installed on the host system. For information on downloading and installing the debugger tools, see Download Debugging Tools for Windows.

Determining the IP Address of the Host Computer

Use one of the following procedures to determine the IP address of the host computer.

1. On the host computer, open a Command Prompt window and enter the following command:

   ```
   ipconfig
   ```

   Make a note of the IPv4 address of the network adapter that you intend to use for debugging.

2. On the target computer, open a Command Prompt window and enter the following command, where

YourIPAddress is the IP address of the host computer:

```
ping -4 <YourIPAddress>
```

Choosing a Port for Network Debugging

Choose a port number that will be used for debugging on both the host and target computers. You can choose any number from 49152 through 65535, the recommended range is 50000 - 50039. The port that you choose will be opened for exclusive access by the debugger running on the host computer. Take care to choose a port number that is not used by any other applications that run on the host computer.

Note The range of port numbers that can be used for network debugging might be limited by your company's network policy. There is no way to tell from the host computer what the limitations are. To determine whether your company's policy limits the range of ports that can be used for network debugging, check with your network administrators.

If you connect several target computers to a single host computer, each connection must have a unique port number. For example, if you connect 100 target computers to a single host computer, you can assign port 50000 to the first connection, port 50001 to the second connection, port 50002 to the third connection, and so on.

Note A different host computer could use the same range of ports (50000 through 50099) to connect to another 100 target computers.

Setting Up the Target Computer

1. Verify that the target computer has a supported network adapter. See these topics for more information.
   - Supported Ethernet NICs for Network Kernel Debugging in Windows 8.1
   - Supported Ethernet NICs for Network Kernel Debugging in Windows 10
2. Connect the supported adapter to a network hub or switch using an appropriate network cable.

> **IMPORTANT**
> Before using BCDEdit to change boot information you may need to temporarily suspend Windows security features such as BitLocker and Secure Boot on the test PC. Re-enable these security features when testing is complete and appropriately manage the test PC, when the security features are disabled.

3. In an elevated Command Prompt window, enter the following commands, where *w.x.y.z* is the IP address of the host computer, and *n* is a port number of your choice:

   ```
   bcdedit /debug on

   bcdedit /dbgsettings net hostip:w.x.y.z port:n
   ```

4. **bcdedit** will display an automatically generated key. Copy the key and store it on a removable storage device like a USB flash drive. You will need the key when you start a debugging session on the host computer.

   Note We strongly recommend that you use an automatically generated key. However, you can create your own key as described later in the "Creating Your Own Key" section.

5. If there is more than one network adapter in the target computer, use Device Manager to determine the PCI bus, device, and function numbers for the adapter you want to use for debugging. Then in an

elevated Command Prompt window, enter the following command, where *b*, *d*, and *f* are the bus number, device number, and function number of the adapter:

```
bcdedit /set "{dbgsettings}" busparams b.d.f
```

6. The target PC will be rebooted after a kernel debugger is attached. This is described in the next section.

Note If you intend to install the Hyper-V role on the target computer, see Setting Up Network Debugging of a Virtual Machine Host.

Caution If your target computer is in a docking station, and you have network debugging enabled for a network adapter that is part of the docking station, do not remove the computer from the docking station. If you need to remove the target computer from the docking station, disable kernel debugging first. To disable kernel debugging on the target computer, open a Command Prompt window as Administrator and enter the command **bcdedit /debug off**. Reboot the target computer.

Starting the Debugging Session

Confirm that the network adapter of the host computer to a network hub or switch using an appropriate network cable.

On the host computer, open WinDbg. On the **File** menu, choose **Kernel Debug**. In the Kernel Debugging dialog box, open the **Net** tab. Enter your port number and key. Click **OK**.

You can also start a session with WinDbg by opening a Command Prompt window and entering the following command, where *n* is your port number and *MyKey* is the key that was automatically generated by **bcdedit** when you set up the target computer:

```
windbg -k net:port=<n>,key=<MyKey>
```

If you are prompted about allowing WinDbg to access the port through the firewall, allow WinDbg to access the port for **all three** of the different network types.

Using KD

On the host computer, open a Command Prompt window. Enter the following command, where*n* is your port number and *MyKey* is the key that was automatically generated by **bcdedit** when you set up the target computer:

```
kd -k net:port=<n>,key=<MyKey>
```

If you are prompted about allowing WinDbg to access the port through the firewall, allow WinDbg to access the port for **all three** of the different network types.

Restarting the Target PC

Once the debugger is connected, and waiting to connect, reboot the target computer. One way to do restart the PC is to use this command, from an administrator's command prompt.

```
shutdown -r -t 0
```

When the target is restarted, the debugger in the host OS should connect.

After connecting to the target on the host, hit break on your debugger and you can start debugging.

Allowing the debugger through the firewall

When you first attempt to establish a network debugging connection, you might be prompted to allow the debugging application (WinDbg or KD) access through the firewall. Client versions of Windows display the prompt, but Server versions of Windows do not display the prompt. You should respond to the prompt by checking the boxes for **all three** network types: domain, private, and public. If you do not get the prompt, or if you did not check the boxes when the prompt was available, you must use Control Panel to allow access through the firewall. Open **Control Panel > System and Security** and click **Allow an app through Windows Firewall**. In the list of applications, locate Windows GUI Symbolic Debugger and Windows Kernel Debugger. Use the check boxes to allow those two applications through the firewall. Restart your debugging application (WinDbg or KD).

Encryption key

To keep the target computer secure, packets that travel between the host and target computers must be encrypted. We strongly recommend that you use an automatically generated encryption key (provided by **bcdedit** when you configure the target computer). Network debugging uses a 256-bit key that is specified as four 64-bit values, in base 36, separated by periods. Each 64-bit value is specified by using up to 13 characters. Valid characters are the letters a through z and the digits 0 through 9. Special characters are not allowed.

To specify your own key, open an elevated Command Prompt window on the target computer. Enter the following command, where *w.x.y.z* is the IP address of the host computer, and *n* is your port number, and *Key* is your key:

```
bcdedit /dbgsettings net hostip:w.x.y.z port:n key:Key
```

The target computer needs to be rebooted anytime the dbgsettings are changed.

Troubleshooting Tips

Debugging application must be allowed through firewall

When you first attempt to establish a network debugging connection, you might be prompted to allow the debugging application (WinDbg or KD) access through the firewall. Client versions of Windows display the prompt, but Server versions of Windows do not display the prompt. You should respond to the prompt by checking the boxes for **all three** network types: domain, private, and public. If you do not get the prompt, or if you did not check the boxes when the prompt was available, you must use Control Panel to allow access through the firewall. Open **Control Panel > System and Security** and click **Allow an app through Windows Firewall**. In the list of applications, locate *Windows GUI Symbolic Debugger* and *Windows Kernel Debugger*. Use the check boxes to allow those two applications through the firewall. Scroll down and click **OK**, to save the firewall changes. Restart the debugger.

Port number must be in range allowed by network policy

The range of port numbers that can be used for network debugging might be limited by your company's network policy. To determine whether your company's policy limits the range of ports that can be used for network debugging, check with your network administrator. On the target computer, open a Command Prompt window as Administrator and enter the command **bcdedit /dbgsettings**. The output will be similar to this.

```
C:\> bcdedit /dbgsettings
key                     XXXXXX.XXXXX.XXXXX.XXXXX
debugtype               NET
hostip                  169.168.1.1
port                    50085
dhcp                    Yes
The operation completed successfully.
```

In the preceding output, the value of port is 50085. If the value of port lies outside the range allowed by your network administrator, enter the following command, where *w.x.y.z* is the IP address of the host computer, and *YourDebugPort* is a port number in the allowed range.

```
bcdedit /dbgsettings net hostip:w.x.y.z port:YourDebugPort
```

After changing the target machine debugger settings, rerun the debugger on the host machine with the new port setting, and then reboot the target computer.

Use Ping to test connectivity

If the debugger does not connect use the ping command on the target PC to verify connectivity.

```
C:\>Ping <HostComputerIPAddress>
```

Note that this may not work if your host computer is not configured to be discoverable on the network, since the firewall may block ping requests, and because of this, you will not get any responses when you ping the host.

How the Debugger Obtains an IP Address for the Target Computer

KDNET on the target computer attempts to use Dynamic Host Configuration Protocol (DHCP) to get a routable IP address for the network adapter that is being used for debugging. If KDNET obtains a DHCP-assigned address, then the target computer can be debugged by host computers located anywhere on the network. If KDNET fails to obtain a DHCP-assigned address, it uses Automatic Private IP Addressing (APIPA) to obtain a local link IP address. Local link IP addresses are not routable, so a host and target cannot use a local link IP address to communicate through a router. In that case, network debugging will work if you plug the host and target computers into the same network hub or switch.

Always specify busparams when setting up KDNET on a physical machine with a PCI based NIC

If you are setting up KDNET on a physical machine with a PCI or PCIe based NIC, you should always specify the busparams for the NIC you want to use for KDNET. To specify the bus parameters, Open Device Manager, and locate the network adapter that you want to use for debugging. Open the property page for the network adapter, and make a note of the bus number, device number, and function number. In an elevated Command Prompt Window, enter the following command, where *b*, *d*, and *f* are the bus, device and function numbers in decimal format:

```
bcdedit /set "{dbgsettings}" busparams b.d.f
```

When the debugger is running on the host machine, and waiting to connect, reboot the target computer, using this command.

```
shutdown -r -t 0
```

Manually delete BCDEdit entries

Manually deleting is not normally required but is provided here as a troubleshooting procedure for unusual situations.

Manually deleting entries is not necessary when using the kdnet utility. For more information, see Setting Up KDNET Network Kernel Debugging Automatically.

When you use bcdedit –deletevalue, you must provide a valid bcd element name. For more information, see BCDEdit /deletevalue.

To manually delete BCDEdit entries, complete these steps.

1. On the target computer, open a Command Prompt window as Administrator.
2. As an example, enter this command to delete the BCDEdit debugging entry for the host IP address.

```
bcdedit -deletevalue {dbgsettings} hostip
```

When you delete the hostip, you need to specify *target*= on the debugger command line.

3. As another example, delete the port entry using this command.

```
bcdedit -deletevalue {dbgsettings} port
```

When you delete the port entry, KDNET will use the default ICANN registered debugger port of 5364.

Hyper-V

Setting up Hyper-V

If you intend to install the Hyper-V role on the target computer, see Setting Up Network Debugging of a Virtual Machine Host.

For information on debugging a hyper-v Virtual Machine (VM), see Setting Up Network Debugging of a Virtual Machine - KDNET.

Enabling KDNET on a hyper-v host that is running VMs with external network connectivity

There is a specific situation, which is not uncommon, which will cause networking in VMs to stop working:

- Hyper-V has been enabled on the PC, an external networking switch has been created and is pointed at a physical NIC in the machine, and VMs have been configured to use that external switch for their networking.
- KDNET is then enabled on the hyper-v host OS using the same physical NIC that is pointed to by the external networking switch, and the host is rebooted.
- All of the VMs that were using the previously configured external switch, lose their network connectivity after the reboot.

This is by design, and happens because KDNET takes exclusive control over the NIC it is configured to use, and the native NDIS miniport for that NIC is not loaded by the OS. When this occurs, the external networking switch can no longer communicate with the native NDIS miniport driver, and will stop working. To work around this situation, do the following:

1. Open the Virtual Switch Manager from Hyper-V Manager, select your existing Virtual Switch, and change the external network NIC to the *Microsoft Kernel Debug Network Adapter* by selecting it from the drop down box and then clicking OK in the Virtual Switch Manager dialog box.
2. After updating your Virtual Switch NIC, shutdown and restart your VMs.

When KDNET debugging is turned off, the same procedure will need to be followed to repoint the external switch back to the native NDIS miniport for the NIC. Otherwise VM connectivity will be lost when the machine is rebooted after debugging is disabled.

IPv6

IPv6 support was added in Windows version 1809.

To use IPv6 with the debugger complete these steps.

1. Ping your <debughostname> and note the IPv6 address that is reported on the Reply from output lines.Use this IPv6 address in place of x:y:z:p:d:q:r:n below.
2. Use BCDEdit to delete any existing ip address values in dbgsettings.

```
bcdedit -deletevalue {dbgsettings} hostip
```

3. Set the IPv6 address of the host. There must not be any spaces in the `hostipv6=s:t:u:v:w:x:y:z` string. is is the network port number to use for this target machine, <YourKey> is the four part security key, and <b.d.f> are the bus device function location numbers for the NIC you want to use for KDNET.

```
bcdedit /dbgsettings net hostipv6:s:t:u:v:w:x:y:z port:<YourPort> key:<YourKey> busparams:<b.d.f>
```

4. Type this command to confirm that the dbgsettings are set properly.

```
C:\> bcdedit /dbgsettings
busparams               0.25.0
key                     2steg4fzbj2sz.23418vzkd4ko3.1g34ou07z4pev.1sp3yo9yz874p
debugtype               NET
hostipv6                  2001:db8:0:0:ff00:0:42:8329
port                    50010
dhcp                    Yes
The operation completed successfully.
```

5. On the host machine use this command to start the debugger.

```
Windbg -k net:port=<yournetworkportnumber>,key=<key_output_from_kdnet>,target=::<YourIPv6Address>
```

6. When the debugger is running on the host machine, and waiting to connect, reboot the target computer.
7. The debugger should connect to the host debugger early during boot. You will know that KDNET is using an IPv6 connection because the IP addresses reported in the connected message will be IPv6 addresses instead of IPv4 addresses.

NOTES

- Every debugger bcd setting that allows the hostip to be specified has a corresponding hostipv6 element. There are three.

IPV4	IPV6	USAGE
hostip	hostipv6	For boot and kernel debugging
targethostip	targethostipv6	Specific to kernel debugging
hypervisorhostip	hypervisorhostipv6	For hyper-v debugging

- If you set the hostipv6 style address for any of those kinds of debugging, it means you want and will get IPv6.
- If you set the hostip style address for any of those kinds of debugging, it means you want and will get IPv4.

- The target will only do IPv4 or IPv6, not both at the same time. Which version of the IP protocol is used is controlled by the target machine dbgsettings. If hostip is set, the target will use IPv4. If hostipv6 is set, the target will use IPv6.
- The host debugger will normally auto select use of IPv4 or IPv6. By default the debugger listens on both an IPv4 socket and an IPv6 socket, and connects automatically on either one to the target machine.
- If you want to force use of IPv6 in the debugger on the host, but you want the debugger to listen for a connection from the target, then you can add, `target=::` to the debugger command line. :: is an IPv6 address of 0.
- If you want to force IPv4 debugging in the debugger on the host, but you want the debugger to listen for a connection from the target, then you can add, `target=0.0.0.0` to the debugger command line. 0.0.0.0 is an IPv4 address of 0.
- If you specify, target= on the debugger command line and use a machine name, the debugger will convert that machine name into an IPv4 address and an IPv6 address, and will attempt to connect on both.
- If you specify, target= on the debugger command line, and use an IP address, if the IP address contains any contains any : characters, the debugger will assume it is an IPv6 address, and will force use of IPv6 for that connection. If the IP address contains any . characters, the debugger will assume it is an IPv4 address, and will force use of IPv4 for that connection.
- If you setup IPv6 on the target, and force use of IPv4 on the debugger command line, you will not get a connection.
- If you setup IPv4 on the target, and force use of IPv6 on the debugger command line, you will also not get a connection.

Related topics

Setting Up KDNET Network Kernel Debugging Automatically

Supported Ethernet NICs for Network Kernel Debugging in Windows 10

Supported Ethernet NICs for Network Kernel Debugging in Windows 8.1

Setting Up Kernel-Mode Debugging over a 1394 Cable Manually

12/5/2018 • 4 minutes to read • Edit Online

IMPORTANT

The 1394 transport is available for use in Windows 10, version 1607 and earlier. It is not available in later versions of Windows. You should transition your projects to other transports, such as KDNET using Ethernet. For more information about that transport, see Setting Up Kernel-Mode Debugging over a Network Cable Manually.

Debugging Tools for Windows supports kernel debugging over a 1394 (Firewire) cable. This topic describes how to set up 1394 debugging manually.

The computer that runs the debugger is called the *host computer*, and the computer being debugged is called the *target computer*. The host and target computers must each have a 1394 adapter and must be running Windows XP or later. The host and target computers do not have to be running the same version of Windows.

Setting Up the Target Computer

1. Connect a 1394 cable to the 1394 controllers that you have chosen for debugging on the host and target computers.

IMPORTANT

Before using BCDEdit to change boot information you may need to temporarily suspend Windows security features such as BitLocker and Secure Boot on the test PC. Re-enable these security features when testing is complete and appropriately manage the test PC, when the security features are disabled.

2. In an elevated Command Prompt window, enter the following commands, where *n* is a channel number of your choice, from 0 through 62:

   bcdedit /debug on

   bcdedit /dbgsettings 1394 channel:*n*

3. If there is more than one 1394 controller on the target computer, you must specify the bus, device, and function numbers of the 1394 controller that you intend to use for debugging. For more information, see Troubleshooting Tips for 1394 Debugging.

4. Do not reboot the target computer yet.

Starting a Debugging Session for the First Time

1. Determine the bitness (32-bit or 64-bit) of Windows running on the host computer.

2. On the host computer, open a version of WinDbg (as Administrator) that matches the bitness of Windows running on the host computer. For example, if the host computer is running a 64-bit version of Windows, open the 64-bit version of WinDbg as Administrator.

3. On the **File** menu, choose **Kernel Debug**. In the Kernel Debugging dialog box, open the **1394** tab. Enter your channel number, and click **OK**.

At this point, the 1394 debug driver gets installed on the host computer. This is why it is important to match the bitness of WinDbg to the bitness of Windows. After the 1394 debug driver is installed, you can use either the 32-bit or 64-bit version of WinDbg for subsequent debugging sessions.

4. Reboot the target computer.

Starting a Debugging Session

Using WinDbg

- On the host computer, open WinDbg. On the **File** menu, choose **Kernel Debug**. In the Kernel Debugging dialog box, open the **1394** tab. Enter your channel number, and click **OK**.

  You can also start a session with WinDbg by entering the following command in a Command Prompt window, where *n* is your channel number:

  windbg /k 1394:channel=*n*

Using KD

- On the host computer, open a Command Prompt window and enter the following command, where *n* is your channel number:

  kd /k 1394:channel=*n*

Using Environment Variables

On the host computer, you can use environment variables to specify the 1394 channel. Then you do not have to specify the channel each time you start a debugging session. To use environment variables to specify the 1394 channel, open a Command Prompt window and enter the following commands, where *n* is your channel number:

- **set _NT_DEBUG_BUS=1394**
- **set _NT_DEBUG_1394_CHANNEL=***n*

To start a debugging session, open a Command Prompt window and enter one of the following commands:

- **kd**
- **windbg**

Additional Information

For complete documentation of the **bcdedit** command and the boot.ini file, see Boot Options for Driver Testing and Debugging in the Windows Driver Kit (WDK) documentation.

Troubleshooting Tips for Debugging over a 1394 Cable

Most 1394 debugging problems are caused by using multiple 1394 controllers in either the host or target computer. Using multiple 1394 controllers in the host computer is not supported. The 1394 debug driver, which runs on the host, can communicate only with the first 1394 controller enumerated in the registry. If you have a 1394 controller built into the motherboard and a separate 1394 card, either remove the card or disable the built-in controller in the BIOS settings of the computer.

The target computer can have multiple 1394 controllers, although this is not recommended. If your target computer has a 1394 controller on the motherboard, use that controller for debugging, if possible. If there is an additional 1394 card, you should remove the card and use the onboard controller. Another solution is to disable the onboard 1394 controller in the BIOS settings of the computer.

If you decide to have multiple 1394 controllers enabled on the target computer, you must specify bus parameters

so that the debugger knows which controller to claim for debugging. To specify the bus parameters, Open Device Manager on the target computer, and locate the 1394 controller that you want to use for debugging. Open the property page for the controller, and make a note of the bus number, device number, and function number. In an elevated Command Prompt Window, enter the following command, where *b*, *d*, and *f* are the bus, device and function numbers in decimal format:

bcdedit -set "{dbgsettings}" busparams *b*.*d*.*f*.

Reboot the target computer.

Related topics

Setting Up Kernel-Mode Debugging Manually

Setting Up Kernel-Mode Debugging over a USB 3.0 Cable Manually

4/24/2019 • 5 minutes to read • Edit Online

Debugging Tools for Windows supports kernel debugging over a USB 3.0 cable. This topic describes how to set up USB 3.0 debugging manually.

The computer that runs the debugger is called the *host computer*, and the computer being debugged is called the *target computer*.

Debugging over a USB 3.0 cable requires the following hardware:

- A USB 3.0 debug cable. This is an A-A crossover cable that has only the USB 3.0 lines and no Vbus.
- On the host computer, an xHCI (USB 3.0) host controller
- On the target computer, an xHCI (USB 3.0) host controller that supports debugging

Setting Up the Target Computer

1. On the target computer, launch the UsbView tool. The UsbView tool is included in Debugging Tools for Windows.
2. In UsbView, locate all of the xHCI host controllers.
3. In UsbView, expand the nodes of the xHCI host controllers. Look for an indication that a port on the host controller supports debugging.

```
[Port1]

Is Port User Connectable:         yes
Is Port Debug Capable:            yes
Companion Port Number:            3
Companion Hub Symbolic Link Name: USB#ROOT_HUB30#5&32bab638&0&0#{...}
Protocols Supported:
 USB 1.1:                         no
 USB 2.0:                         no
 USB 3.0:                         yes
```

4. Make a note of the bus, device, and function numbers for the xHCI controller that you intend to use for debugging. UsbView displays these number. In the following example, the bus number is 48, the device number is 0, and the function number is 0.

```
USB xHCI Compliant Host Controller
...
DriverKey: {36fc9e60-c465-11cf-8056-444553540000}\0020
...
Bus.Device.Function (in decimal): 48.0.0
```

5. After you have identified an xHCI controller that supports debugging, the next step is to locate the physical USB connector that is associated with a port on the xHCI controller. To find the physical connector, plug any USB 3.0 device into any USB connector on the target computer. Refresh UsbView to see where your device is located. If UsbView shows your device connected to your chosen xHCI host controller, then you have

found a physical USB connector that you can use for USB 3.0 debugging.

> **IMPORTANT**
> Before using bcdedit to change boot information you may need to temporarily suspend Windows security features such as BitLocker and Secure Boot on the test PC. You can re-enable Secure Boot once you're done debugging and you've disabled kernel debugging.

6. On the target computer, open a Command Prompt window as Administrator, and enter these commands:

   - **bcdedit /debug on**
   - **bcdedit /dbgsettings usb targetname:***TargetName*

   where *TargetName* is a name that you create for the target computer. Note that *TargetName* does not have to be the official name of the target computer; it can be any string that you create as long as it meets these restrictions:

   - The string must not contain "debug" anywhere in the *TargetName* in any combination of upper or lower case. For example if you use "DeBuG" or "DEBUG" anywhere in your targetname, debugging will not work correctly.
   - The only characters in the string are the hyphen (-), the underscore(_), the digits 0 through 9, and the letters A through Z (upper or lower case).
   - The maximum length of the string is 24 characters.
7. If you have more than one USB host controller on the target computer, enter this command:

   bcdedit /set "{dbgsettings}" busparams *b.d.f*

   where *b*, *d*, and *f* are the bus, device, and function numbers for the USB host controller that you intend to use for debugging. The bus, device, and function numbers must be in decimal format.

   Example:

   bcdedit /set "{dbgsettings}" busparams 48.0.0
8. Reboot the target computer.

Starting a Debugging Session for the First Time

1. Connect a Universal Serial Bus (USB) 3.0 debug cable to the USB 3.0 ports that you have chosen for debugging on the host and target computers.
2. Determine the bitness (32-bit or 64-bit) of Windows running on the host computer.
3. On the host computer, open a version of WinDbg (as Administrator) that matches the bitness of Windows running on the host computer. For example, if the host computer is running a 64-bit version of Windows, open the 64-bit version of WinDbg as Administrator.
4. On the **File** menu, choose **Kernel Debug**. In the Kernel Debugging dialog box, open the **USB** tab. Enter the target name that you created when you set up the target computer. Click **OK**.

At this point, the USB debug driver gets installed on the host computer. This is why it is important to match the bitness of WinDbg to the bitness of Windows. After the USB debug driver is installed, you can use either the 32-bit or 64-bit version of WinDbg for subsequent debugging sessions.

Starting a Debugging Session

Using WinDbg

On the host computer, open WinDbg. On the **File** menu, choose **Kernel Debug**. In the Kernel Debugging dialog

box, open the **USB** tab. Enter the target name that you created when you set up the target computer. Click **OK**.

You can also start a session with WinDbg by entering the following command in a Command Prompt window, where *TargetName* is the target name you created when you set up the target computer:

windbg /k usb:targetname=*TargetName*

Using KD

On the host computer, open a Command Prompt window and enter the following command, where *TargetName* is the target name you created when you set up the target computer:

kd /k usb:targetname=*TargetName*

Troubleshooting tips for debugging over USB 3.0

In some cases, power transitions can interfere with debugging over USB 3.0. If you have this problem, disable selective suspend for the xHCI host controller (and its root hub) that you are using for debugging.

1. In Device Manager, navigate to the node for the xHCI host controller. Right click the node, and choose **Properties**. If there is a **Power Management** tab, open the tab, and clear the **Allow the computer to turn off this device to save power** check box.
2. In Device Manager, navigate to the node for the root hub of the xHCI host controller. Right click the node, and choose **Properties**. If there is a **Power Management** tab, open the tab, and clear the **Allow the computer to turn off this device to save power** check box.

When you have finished using the xHCI host controller for debugging, enable selective suspend for the xHCI host controller.

Related topics

Setting Up Kernel-Mode Debugging Manually

Setting Up Kernel-Mode Debugging over a USB 2.0 Cable Manually

4/24/2019 • 7 minutes to read • Edit Online

Debugging Tools for Windows supports kernel debugging over a USB 2.0 cable. This topic describes how to set up USB 2.0 debugging manually.

The computer that runs the debugger is called the *host computer*, and the computer being debugged is called the *target computer*.

Debugging over a USB 2.0 cable requires the following hardware:

- A USB 2.0 debug cable. This cable is not a standard USB 2.0 cable because it has an extra hardware component that makes it compatible with the USB2 Debug Device Functional Specification. You can find these cables with an Internet search for the term *USB 2.0 debug cable*.
- On the host computer, an EHCI (USB 2.0) host controller
- On the target computer, an EHCI (USB 2.0) host controller that supports debugging

Setting Up the Target Computer

1. On the target computer, launch the UsbView tool. The UsbView tool is included in Debugging Tools for Windows.
2. In UsbView, locate all of the host controllers that are compatible with the EHCI specification. For example, you could look for controllers that are listed as Enhanced.
3. In UsbView, expand the nodes of the EHCI host controllers. Look for an indication that a host controller supports debugging, and look for the number of the debug port. For example, UsbView displays this output for an EHCI host controller that supports debugging on port 1.

   ```
   Xxx xxx xxx USB2 Enhanced Host Controller - 293A
   ...
   Debug Port Number:  1
   Bus.Device.Function (in decimal): 0.29.7
   ```

   Note Many EHCI host controllers support debugging on port 1, but some EHCI host controllers support debugging on port 2.

4. Make a note of the bus, device, and function numbers for the EHCI controller that you intend to use for debugging. UsbView displays these number. In the preceding example, the bus number is 0, the device number is 29, and the function number is 7.
5. After you have identified the EHCI controller and the port number that supports debugging, the next step is to locate the physical USB connector that is associated with the correct port number. To find the physical connector, plug any USB 2.0 device into any USB connector on the target computer. Refresh UsbView to see where your device is located. If UsbView shows your device connected to the EHCI host controller and port that you identified as the debug port, then you have found a physical USB connector that you can use for debugging. It could be that there is no external physical USB connector that is associated with a debug port on an EHCI controller. In that case, you can look for a physical USB connector inside the computer. Perform the same steps to determine whether the internal USB connector is suitable for kernel debugging.

If you cannot find a physical USB connector (either external or internal) that is associated with a debug port, then you cannot use the computer as a target for debugging over a USB 2.0 cable.

Note See this remark for an exception.

> **IMPORTANT**
> Before using bcdedit to change boot information you may need to temporarily suspend Windows security features such as BitLocker and Secure Boot on the test PC. You can re-enable Secure Boot once you're done debugging and you've disabled kernel debugging.

6. On the target computer, open a Command Prompt window as Administrator, and enter these commands:

   - **bcdedit /debug on**
   - **bcdedit /dbgsettings usb targetname:***TargetName*

   where *TargetName* is a name that you create for the target computer. Note that *TargetName* does not have to be the official name of the target computer; it can be any string that you create as long as it meets these restrictions:

   - The string must not contain "debug" anywhere in the *TargetName* in any combination of upper or lower case. For example if you use "DeBuG" or "DEBUG" anywhere in your targetname, debugging will not work correctly.
   - The only characters in the string are the hyphen (-), the underscore(_), the digits 0 through 9, and the letters A through Z (upper or lower case).
   - The maximum length of the string is 24 characters.
7. If there is more than one USB host controller on the target computer, enter this command:

   Windows 7 or later
   bcdedit /set "{dbgsettings}" busparams *b.d.f*

   where *b*, *d*, and *f* are the bus, device, and function numbers for the host controller. The bus, device, and function numbers must be in decimal format (for example, **busparams 0.29.7**).

   Windows Vista
   bcdedit /set "{current}" loadoptions busparams=*f.d.f*

   where *b*, *d*, and *f* are the bus, device, and function numbers for the host controller. The bus, device, and function numbers must be in hexadecimal format (for example, **busparams=0.1D.7**).
8. Reboot the target computer.

Setting Up the Host Computer

1. Verify that the host computer is not configured to be the target of USB debugging. (If necessary, open a Command Prompt window as Administrator, enter **bcdedit /debug off**, and reboot.)
2. On the host computer, use UsbView to find the EHCI host controllers and ports that support debugging. If possible, plug one end of the USB 2.0 debug cable into an EHCI port (on the host computer) that does not support debugging. Otherwise, plug the cable into any EHCI port on the host computer.
3. Plug the other end of the USB 2.0 debug cable into the connector that you identified previously on the target computer.

Starting a Debugging Session for the First Time

1. Determine the bitness (32-bit or 64-bit) of Windows running on the host computer.
2. On the host computer, open a version of WinDbg (as Administrator) that matches the bitness of Windows

running on the host computer. For example, if the host computer is running a 64-bit version of Windows, open the 64-bit version of WinDbg as Administrator.

3. On the **File** menu, choose **Kernel Debug**. In the Kernel Debugging dialog box, open the **USB** tab. Enter the target name that you created when you set up the target computer. Click **OK**.

At this point, the USB debug driver gets installed on the host computer. This is why it is important to match the bitness of WinDbg to the bitness of Windows. After the USB debug driver is installed, you can use either the 32-bit or 64-bit version of WinDbg for subsequent debugging sessions.

Note The USB 2.0 debug cable is actually two cables with a dongle in the middle. The direction of the dongle is important; one side powers the device, and the other side does not. If USB debugging doesn't work, try swapping the direction of the dongle. That is, unplug both cables from the dongle, and swap the sides that the cables are connected to.

Starting a Debugging Session

Using WinDbg

On the host computer, open WinDbg. On the **File** menu, choose **Kernel Debug**. In the Kernel Debugging dialog box, open the **USB** tab. Enter the target name that you created when you set up the target computer. Click **OK**.

You can also start a session with WinDbg by entering the following command in a Command Prompt window, where *TargetName* is the target name you created when you set up the target computer:

windbg /k usb:targetname=*TargetName*

Using KD

On the host computer, open a Command Prompt window and enter the following command, where *TargetName* is the target name you created when you set up the target computer:

kd /k usb:targetname=*TargetName*

What if USBView shows a debug-capable port, but does not show the port mapped to any physical connector?

On some computers, USBView shows a debug-capable port, but does not show the port mapped to any physical USB connector. For example, USBView might show port 2 as the debug port number for an eHCI controller.

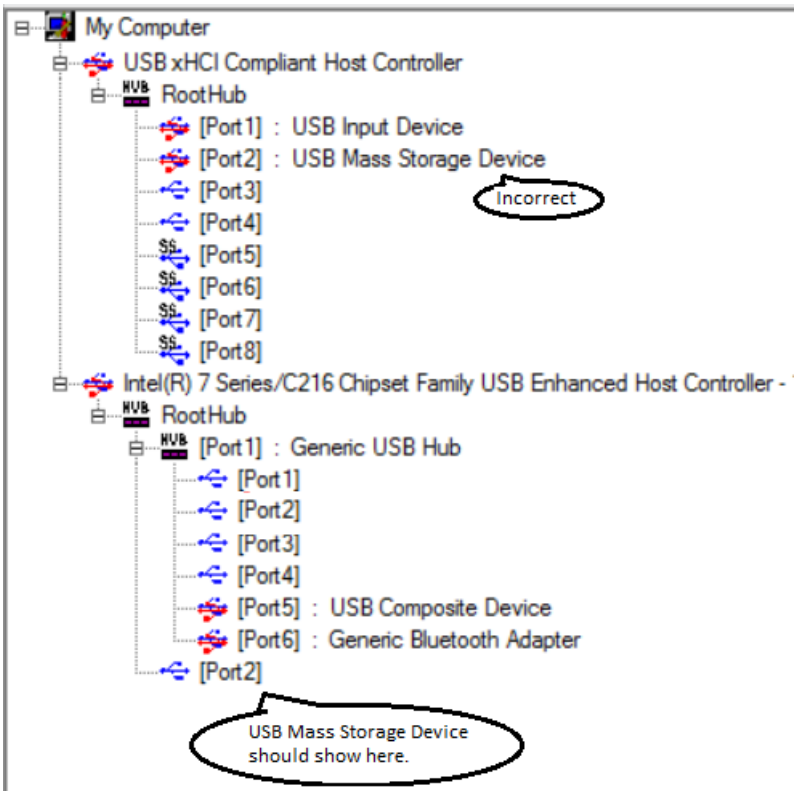
```
... USB Enhanced Host Controller ...
...
Debug Port Number:  2
Bus.Device.Function (in decimal): 0.29.0
```

Also, when you use USBView to look at the individual port, it is listed as debug-capable.

```
[Port 2]
Is Port User Connectiable: Yes
Is Port Debug Capable: Yes
...
Protocols Supported
  USB 1.1      yes
  USB 2.0      yes
  USB 3.0      no
```

But when you plug in a USB 2.0 device (like a flash drive) to all the USB connectors on the computer, USBView never show your device connected to the debug-capable port (port 2 in this example). USBView might show the external connector mapped to a port of an xHCI controller when in fact the external connector is mapped to the

debug-capable port of the eHCI controller.



In a case like this, you might still be able to establish kernel-mode debugging over a USB 2.0 cable. In the example given here, you would plug your USB 2.0 debug cable into the connector that shows as being mapped to Port 2 of the xHCI controller. Then you would set your bus parameters to the bus, device, and function numbers of the eHCI controller (in this example, 0.29.0).

bcdedit /set "{dbgsettings}" busparams 0.29.0

Additional Support

For troubleshooting tips and detailed instructions on setting up kernel debugging over USB, see Setting Up Kernel Debugging with USB 2.0.

Related topics

Setting Up Kernel-Mode Debugging Manually

Setting Up Kernel-Mode Debugging over a Serial Cable Manually

12/5/2018 • 4 minutes to read • Edit Online

Debugging Tools for Windows supports kernel debugging over a null-modem cable. Null-modem cables are serial cables that have been configured to send data between two serial ports. They are available at most computer stores. Do not confuse null-modem cables with standard serial cables. Standard serial cables do not connect serial ports to each other. For information about how null-modem cables are wired, see Null-Modem Cable Wiring.

The computer that runs the debugger is called the *host computer*, and the computer being debugged is called the *target computer*.

Setting Up the Target Computer

> **IMPORTANT**
> Before using bcdedit to change boot information you may need to temporarily suspend Windows security features such as BitLocker and Secure Boot on the test PC. You can re-enable Secure Boot once you're done debugging and you've disabled kernel debugging.

1. On the target computer, open a Command Prompt window as Administrator, and enter the following commands, where *n* is the number of the COM port used for debugging on the target computer, and *rate* is the baud rate used for debugging:

   bcdedit /debug on

   bcdedit /dbgsettings serial debugport:*n* **baudrate:***rate*

   Note The baud rate must be the same on the host computer and target computer. The recommended rate is 115200.

2. Reboot the target computer.

Starting the Debugging Session

Connect the null-modem cable to the COM ports that you have chosen for debugging on the host and target computers.

Using WinDbg

On the host computer, open WinDbg. On the **File** menu, choose **Kernel Debug**. In the Kernel Debugging dialog box, open the **COM** tab. In the **Baud rate** box, enter the rate you have chosen for debugging. In the **Port** box, enter COM*n* where *n* is the COM port number you have chosen for debugging on the host computer. Click **OK**.

You can also start a session with WinDbg by entering the following command in a Command Prompt window; *n* is the number of the COM port used for debugging on the host computer, and *rate* is the baud rate used for debugging:

windbg -k com:port=COM*n***,baud=***rate*

Using KD

On the host computer, open a Command Prompt window, and enter the following command, where*n* is the number of the COM port used for debugging on the host computer, and *rate* is the baud rate used for debugging:

kd -k com:port=COM*n*,**baud=***rate*

Using Environment Variables

On the host computer, you can use environment variables to specify the COM port and the baud rate. Then you do not have to specify the port and baud rate each time you start a debugging session. To use environment variables to specify the COM port and baud rate, open a Command Prompt window and enter the following commands, where *n* is the number of the COM port used for debugging on the host computer, and *rate* is the baud rate used for debugging:

- **set _NT_DEBUG_PORT=COM***n*
- **set _NT_DEBUG_BAUD_RATE=***rate*

To start a debugging session, open a Command Prompt window, and enter one of the following commands:

- **kd**
- **windbg**

Troubleshooting Tips for Debugging over a Serial Cable

Specify correct COM port on both host and target

Determine the numbers of the COM ports you are using for debugging on the host and target computers. For example, suppose you have your null-modem cable connected to COM1 on the host computer and COM2 on the target computer.

On the target computer, open a Command Prompt window as Administrator, and enter **bcdedit /dbgsettings**. If you are using COM2 on the target computer, the output of **bcdedit** should show `debugport 2`.

On the host computer, specify the correct COM port when you start the debugger or when you set environment variables. If you are using COM1 on the host computer, use one of the following methods to specify the COM port.

- In WinDbg, in the Kernel Debugging dialog box, enter COM1 in the **Port** box.
- **windbg -k com:port=COM1, ...**
- **kd -k com:port=COM1, ...**
- **set _NT_DEBUG_PORT=COM1**

Baud rate must be the same on host and target

The baud rate used for debugging over a serial cable must be set to the same value on the host and target computers. For example, suppose you have chosen a baud rate of 115200.

On the target computer, open a Command Prompt window as Administrator, and enter **bcdedit /dbgsettings**. The output of **bcdedit** should show `baudrate 115200`.

On the host computer, specify the correct baud rate when you start the debugger or when you set environment variables. Use one of the following methods to specify a baud rate of 115200.

- In WinDbg, in the Kernel Debugging dialog box, enter 115200 in the **Baud rate** box.
- **windbg -k ..., baud=115200**
- **kd -k ..., baud=115200**
- **set _NT_DEBUG_BAUD_RATE=115200**

Null Modem Cable Wiring

The following tables show how null-modem cables are wired.

9-pin connector

CONNECTOR 1	CONNECTOR 2	SIGNALS
2	3	Tx - Rx
3	2	Rx - Tx
7	8	RTS - CTS
8	7	CTS - RTS
4	1+6	DTR - (CD+DSR)
1+6	4	(CD+DSR) - DTR
5	5	Signal ground

25-pin connector

CONNECTOR 1	CONNECTOR 2	SIGNALS
2	3	Tx - Rx
3	2	Rx - Tx
4	5	RTS - CTS
5	4	CTS - RTS
6	20	DSR - DTR
20	6	DTR - DSR
7	7	Signal ground

Signal Abbreviations

ABBREVIATION	SIGNAL
Tx	Transmit data
Rx	Receive data
RTS	Request to send
CTS	Clear to send
DTR	Data terminal ready

ABBREVIATION	SIGNAL
DSR	Data set ready
CD	Carrier detect

Additional Information

For complete documentation of the **bcdedit** command, see Boot Options for Driver Testing and Debugging in the Windows Driver Kit (WDK) documentation.

Related topics

Setting Up Kernel-Mode Debugging Manually

Setting Up Kernel-Mode Debugging using Serial over USB Manually

12/5/2018 • 4 minutes to read • Edit Online

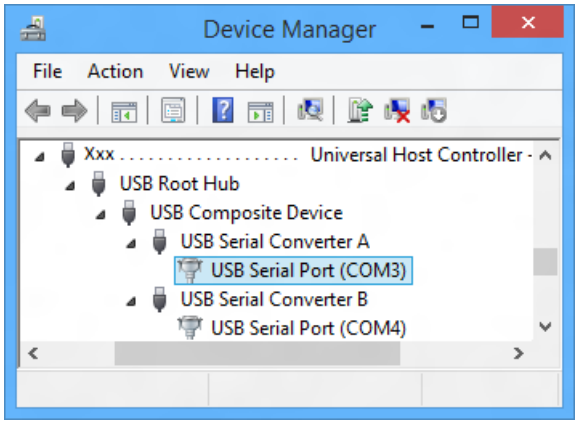
The Sharks Cove development board supports serial debugging over a USB cable.

The computer that runs the debugger is called the *host computer*, and the computer being debugged is called the *target computer*. In this topic, the Sharks Cove board is the target computer.

Setting up a Host Computer for debugging the Sharks Cove board

1. On the host computer, open Device Manager. On the **View** menu, choose **Devices by Type**.
2. On the Sharks Cove board, locate the debug connector. This is the micro USB connector shown in the following picture.

3. Use a USB 2.0 cable to connect the host computer to the debug connector on the Sharks cove board.
4. On the host computer, in Device Manager, two COM ports will get enumerated. Select the lowest numbered COM port. On the **View** menu, choose **Devices by Connection**. Verify that the COM port is listed under one of the USB host controllers.



Make a note of the COM port number. This is the lowest COM port number that shows under the USB host controller node. For example, in the preceding screen shot, the lowest COM port number under the USB host controller is COM3. You will need this COM port number later when you start a debugging session. If the driver is not already installed for the COM port, right click the COM port node, and choose **Update Driver**. Then select **Search automatically for updated driver software**. You will need an internet connection for this.

Setting Up the Sharks Cove Board as the Target Computer

> **IMPORTANT**
> Before using BCDEdit to change boot information you may need to temporarily suspend Windows security features such as BitLocker and Secure Boot on the test PC. Re-enable these security features when testing is complete and appropriately manage the test PC, when the security features are disabled.

1. On the target computer (Sharks Cove board), open a Command Prompt window as Administrator, and enter these commands:

   bcdedit /debug on

   bcdedit /dbgsettings serial debugport:1 baudrate:115200

2. Reboot the target computer.

Starting the Debugging Session

Using WinDbg

On the host computer, open WinDbg. On the **File** menu, choose **Kernel Debug**. In the Kernel Debugging dialog box, open the **COM** tab. In the **Baud rate** box, enter 115200. In the **Port** box, enter COM*n* where *n* is the COM port number you noted previously. Click **OK**.

You can also start a session with WinDbg by entering the following command in a Command Prompt window; *n* is the number of the COM port that you noted on the host computer:

windbg -k com:port=COM*n*,baud=115200

Using KD

On the host computer, open a Command Prompt window, and enter the following command, where*n* is the COM port number you noted previously:

kd -k com:port=COM*n*,baud=115200

Using Environment Variables

On the host computer, you can use environment variables to specify the COM port and the baud rate. Then you do not have to specify the port and baud rate each time you start a debugging session. To use environment variables to specify the COM port and baud rate, open a Command Prompt window and enter the following commands, where *n* is the number COM port number you noted previously:

- **set _NT_DEBUG_PORT=COM*n***
- **set _NT_DEBUG_BAUD_RATE=115200**

To start a debugging session, open a Command Prompt window, and enter one of the following commands:

- **kd**
- **windbg**

Troubleshooting Tips for Serial Debugging over a USB Cable

Specify correct COM port on both host and target

On the target computer (Sharks Cove board), verify that you are using COM1 for debugging. Open a Command Prompt window as Administrator, and enter **bcdedit /dbgsettings**. The output of **bcdedit** should show

`debugport 1`.

On the host computer, specify the correct COM port when you start the debugger or when you set environment variables. This is the lowest numbered COM port that was enumerated under the USB host controller in Device Manager. For example, if COM3 is the desired port, use one of the following methods to specify the COM port.

- In WinDbg, in the Kernel Debugging dialog box, enter COM3 in the **Port** box.
- **windbg -k com:port=COM3, ...**
- **kd -k com:port=COM3, ...**
- **set _NT_DEBUG_PORT=COM3**

Baud rate must be the same on host and target

The baud rate must be 115200 on both the host and target computers.

On the target computer (Sharks Cove board), open a Command Prompt window as Administrator, and enter **bcdedit /dbgsettings**. The output of **bcdedit** should show `baudrate 115200`.

On the host computer, specify the correct baud rate when you start the debugger or when you set environment variables. Use one of the following methods to specify a baud rate of 115200.

- In WinDbg, in the Kernel Debugging dialog box, enter 115200 in the **Baud rate** box.
- **windbg -k ..., baud=115200**
- **kd -k ..., baud=115200**
- **set _NT_DEBUG_BAUD_RATE=115200**

Additional Information

For complete documentation of the **bcdedit** command, see Boot Options for Driver Testing and Debugging in the Windows Driver Kit (WDK) documentation.

Related topics

Setting Up Kernel-Mode Debugging Manually

Setting Up Kernel-Mode Debugging of a Virtual Machine Manually using a Virtual COM Port

4/24/2019 • 5 minutes to read • Edit Online

Debugging Tools for Windows supports kernel debugging of a virtual machine. The virtual machine can be located on the same physical computer as the debugger or on a different computer that is connected to the same network. This topic describes how to set up debugging of a virtual machine manually using a virtual COM Port via KDCOM.

Using KDNET virtual networking is a faster option and is recommended. For more information, see Setting Up Network Debugging of a Virtual Machine with KDNET.

Setting Up the Target Virtual Machine

The computer that runs the debugger is called the *host computer*, and the virtual machine being debugged is called the *target virtual machine*.

> **IMPORTANT**
> Before using BCDEdit to change boot information you may need to temporarily suspend Windows security features such as BitLocker and Secure Boot on the test PC. Re-enable these security features when testing is complete and appropriately manage the test PC, when the security features are disabled.

1. In the virtual machine, in an elevated Command Prompt window, enter the following commands.

   bcdedit /debug on

   bcdedit /dbgsettings serial debugport:*n* **baudrate:115200**

   where *n* is the number of a COM port on the virtual machine.

2. In the virtual machine, configure the COM port to map to a named pipe. The debugger will connect through this pipe. For more information about how to create this pipe, see your virtual machine's documentation.

3. Start the debugger in elevated mode, for example from an administrator command prompt. The debugger must be running in elevated mode when debugging a VM over a serial pipe. Once the debugger is attached and running, reboot the target VM.

Starting the Debugging Session Using WinDbg

On the host computer, open WinDbg as an Administrator. The debugger must be running in elevated mode when debugging a VM over a serial pipe. On the **File** menu, choose **Kernel Debug**. In the Kernel Debugging dialog box, open the **COM** tab. Check the **Pipe** box, and check the **Reconnect** box. For **Baud Rate**, enter 115200. For **Resets**, enter 0.

If the debugger is running on the same computer as the virtual machine, enter the following for **Port**.

\\.\pipe*PipeName*.

If the debugger is running on a different computer from the virtual machine, enter the following for **Port**.

*****VMHost*\pipe***PipeName*

Click **OK**.

You can also start WinDbg at the command line. If the debugger is running on the same physical computer as the virtual machine, enter the following command in a Command Prompt window.

windbg -k com:pipe,port=\\.\pipe*PipeName***,resets=0,reconnect**

If the debugger is running on a different physical computer from the virtual machine, enter the following command in a Command Prompt window.

windbg -k com:pipe,port=*VMHost***\pipe***PipeName***,resets=0,reconnect**

Starting the Debugging Session Using KD

To debug a virtual machine that is running on the same physical computer as the debugger, enter the following command in an *elevated* Command Prompt window.

kd -k com:pipe,port=\\.\pipe*PipeName***,resets=0,reconnect**

To debug a virtual machine that is running on a different physical computer from the debugger, enter the following command in a Command Prompt window.

kd -k com:pipe,port=*VMHost***\pipe***PipeName***,resets=0,reconnect**

Parameters

VMHost
Specifies the name of the computer that the virtual machine is running on.

PipeName
Specifies the name of the pipe that you created on the virtual machine.

resets=0
Specifies that an unlimited number of reset packets can be sent to the target when the host and target are synchronizing. Use the **resets=0** parameter for Microsoft Virtual PC and other virtual machines whose pipes drop excess bytes. Do not use this parameter for VMware or other virtual machines whose pipes do not drop all excess bytes.

reconnect
Causes the debugger to automatically disconnect and reconnect the pipe if a read/write failure occurs. Additionally, if the debugger does not find the named pipe when the debugger is started, the *reconnect* parameter causes the debugger to wait for a pipe that is named *PipeName* to appear. Use *reconnect* for Virtual PC and other virtual machines that destroy and re-create their pipes during a computer restart. Do not use this parameter for VMware or other virtual machines that preserve their pipes during a computer restart.

For more information about additional command-line options, see **KD Command-Line Options** or **WinDbg Command-Line Options**.

Generation 2 Virtual Machines

By default, COM ports are not presented in generation 2 virtual machines. You can add COM ports through PowerShell or WMI. For the COM ports to be displayed in the Hyper-V Manager console, they must be created with a path.

To enable kernel debugging using a COM port on a generation 2 virtual machine, follow these steps:

1. Disable Secure Boot by entering this PowerShell command:

Set-VMFirmware –Vmname *VmName* **–EnableSecureBoot Off**

where *VmName* is the name of your virtual machine.

2. Add a COM port to the virtual machine by entering this PowerShell command:

   Set-VMComPort –VMName *VmName* **1 \\.\pipe***PipeName*

   For example, the following command configures the first COM port on virtual machine TestVM to connect to named pipe TestPipe on the local computer.

   Set-VMComPort –VMName TestVM 1 \\.\pipe\TestPipe

3. Once the debugger is attached and running, stop and cold start the VM to activate the COM ports in the VM. The emulated UARTS aren't available for debugging unless at least one is actually configured with a pipe name and they cannot be hot-added.

4. Re-enable secure boot, once you are done updating the configuration changes.

For more information about Generation 2 VMs, see Generation 2 Virtual Machine Overview.

Remarks

If the target computer has stopped responding, the target computer is still stopped because of an earlier kernel debugging action, or you used the **-b** command-line option, the debugger breaks into the target computer immediately.

Otherwise, the target computer continues running until the debugger orders it to break.

Troubleshooting Firewalls and Network Access Issues

Your debugger (WinDbg or KD) must have access through the firewall. This can even be the case for virtual serial ports that are supported by network adapters.

If you are prompted by Windows to turn off the firewall when the debugger is loaded, select all three boxes.

Depending on the specifics of the VM in use, you may need to change the network settings for your virtual machines to bridge them to the Microsoft Kernel Network Debug Adapter. Otherwise, the virtual machines will not have access to the network.

Windows Firewall

You can use Control Panel to allow access through the Windows firewall. Open Control Panel > System and Security, and select Allow an app through Windows Firewall. In the list of applications, locate *Windows GUI Symbolic Debugger* and *Windows Kernel Debugger*. Use the check boxes to allow those two applications through the firewall. Restart your debugging application (WinDbg or KD).

Third Party VMs

VMWare If you restart the virtual machine by using the VMWare facilities (for example, the reset button), exit WinDbg, and then restart WinDbg to continue debugging. During virtual machine debugging, VMWare often consumes 100% of the CPU.

Related topics

Setting Up Network Debugging of a Virtual Machine with KDNET

Setting Up Kernel-Mode Debugging Manually

Setting Up Network Debugging of a Virtual Machine - KDNET

1/3/2019 • 5 minutes to read • Edit Online

This topic describes how to configure a kernel debugging connection to a Hyper-V virtual machine (VM).

Hyper-V Virtual Machine Setup

To debug a Gen 2 Hyper-V Virtual Machine (VM) complete the following steps.

1. Create a VM with Windows installed

For information on how to create a VM, see Create a Virtual Machine with Hyper-V.

2. Define an external virtual switch

To communicate with the VM a virtual external network switch can be used. For information on how to create external network switch, see Create a virtual network.

When the external network switch is configured the following options must be set.

OPTION	VALUE
Connection Type	External Network
Allow management operating system to share this network adapter	Enabled
VLAN ID	Disabled

3. Disable Secure Boot

To allow the kdnet utility to update BCDEdit boot settings, temporarily disable secure boot on the virtual machine by following these steps.

1. Load the Hyper-V manager and select the properties for your VM.
2. Click on the **Security** settings.
3. Un-check the **Enable Secure Boot** checkbox.
4. Click **OK** to save the settings.

You can re-enable Secure Boot once you're done debugging and you've disabled kernel debugging on the target VM.

4. Install the Debugging Tools for Windows

The debugging tools are used for the debugger and the kdnet utility and must be installed. For information on how to download and install the debugging tools, see Download Debugging Tools for Windows.

Setting Up Network Debugging of a Virtual Machine - KDNET

Record the Host IP Address

To run the host debugger on the same PC as the target virtual machine, follow these steps.

1. In the host computer OS, open a Command Prompt window and enter *IPConfig* to display the IP configuration.
2. In the command output, locate the Ethernet adapter that you configured as the External Virtual Switch.

```
...

Ethernet adapter vEthernet (External Virtual Switch):

...

IPv4 Address. . . . . . . . . . . : <YourHostIPAddress>

...
```

3. Record the IPv4 address of the External Virtual Switch that will be used as the host address for debugging.
4. To confirm connectivity between the target and the host computer, open an elevated command prompt window on the target computer, and enter the following command, where *YourHostIPAddress* is the IP address of the host computer.

```
ping -4 <YourHostIPAddress>
```

Setting Up the VM Target Computer

Use the kdnet.exe utility to automatically configure the debugger settings on the target PC, by following these steps.

1. Locate the WDK *kdnet.exe* and *VerifiedNICList.xml* files. By default they are located here.

```
C:\Program Files (x86)\Windows Kits\10\Debuggers\x64
```

> **NOTE**
> These directions assumes that both PCs are running a 64 bit version of Windows on both the target and host. If that is not the case, the best approach is to run the same "bitness" of tools on the host that the target is running. For example, if the target is running 32 bit Windows, run a 32 version of the debugger on the host. For more information, see Choosing the 32-Bit or 64-Bit Debugging Tools.

4. To allow the long key that is used to be cut and pasted, enable enhanced session support. In the VM window, from the **View** pull down menu, enable *Enhanced session*.
5. On the target VM computer, create a C:\KDNET directory and copy the *kdnet.exe* and *VerifiedNICList.xml* files to that directory.
6. On the target computer, open a Command Prompt window as Administrator. Enter this command to verify that the target computer has a supported network adapter.

```
C:\KDNET>kdnet

Network debugging is supported on the following NICs:
busparams=0.25.0, Intel(R) 82579LM Gigabit Network Connection, KDNET is running on this NIC.kdnet.exe
```

7. Type this command to set the IP address of the host system and generated a unique connection key. Use the IP address of the host system you recorded earlier. Pick a unique port address for each target/host pair that you work with, with in the range 50000-50039. For this example, we will select 50005.

```
C:\>kdnet <YourIPAddress> <YourDebugPort>

Enabling network debugging on Microsoft Hypervisor Virtual Machine.
Key=3u8smyv477z20.2owh9gl90gbxx.3sfsihzgq7di4.nh8ugnmzb4l7

To debug this vm, run the following command on your debugger host machine.
windbg -k net:port=50005,key=3u8smyv477z20.2owh9gl90gbxx.3sfsihzgq7di4.nh8ugnmzb4l7

Then restart this VM by running shutdown -r -t 0 from this command prompt.
```

8. Use CRTL+C to copy the provided windbg output into the command buffer. Doing this avoids attempting to write down the long key value that is returned.
9. Re-enable BitLocker and secure boot when you're done configuring the debugger settings.
10. Because a VM with enhanced session support can timeout when it is left in a breakpoint, disable *Enhanced session* support using the **View** pull down menu in the VM.
11. The VM will be restarted after the debugger is loaded and running. This process is described next.

Starting the Debugging Session

2. To connect to the target PC, use CTRL+V to paste in the main OS command window the windbg string that was returned by kdnet that you copied earlier.

```
C:\Debuggers\windbg -k net:port=<YourDebugPort>,key=<YourKey>
```

When you first attempt to establish a network debugging connection, you might be prompted to allow the debugging application (WinDbg or KD) access through the firewall. You should respond to the prompt by checking the boxes for **all three** network types: domain, private, and public.

Restarting the Target PC

Once the debugger is connected, reboot the target computer. To force the VM to completely restart, use this command, from an administrator's command prompt.

```
shutdown -r -t 0
```

When the target virtual machine is restarted, the debugger in the host OS should connect.

After connecting to the VM, hit break on your debugger and you can start debugging.

Troubleshooting KDNET Virtual Machine Network Debugging

If the debugger does not connect, use the ping command from the target VM to verify connectivity.

```
C:\>Ping <HostComputerIPAddress>
```

Something didn't work right and I'm not sure what...

- Ensure you've let WinDbg through your firewall.
- Confirm that you are using a unique Key that was generated by BCDEdit or kdnet.

My VMs don't have network connectivity

- Open Virtual Switch Manager from Hyper-V Manager, select your existing Virtual Switch, and change the external network NIC to the Microsoft Kernel Debug Network Adapter by selecting it from the drop down box and then clicking OK in the Virtual Switch Manager dialog box. After updating your Virtual Switch NIC, make sure to then shutdown and restart your VMs.

Sequence to add Hyper-V role to a Windows PC

If your target computer is a virtual machine host, you can set up network debugging and still have network access for the virtual machines.

Suppose you want to set up network debugging in the following situation.

- The target computer has a single network interface card.
- You intend to install the Hyper-V role on the target computer.
- You intend to create one or more virtual machines on the target computer.

The best approach is to set up network debugging on the target computer before you install the Hyper-V role. Then the virtual machines will have access to the network.

If you decide to set up network debugging after the Hyper-V role has been installed on the target computer, you must change the network settings for your virtual machines to bridge them to the Microsoft Kernel Network Debug Adapter. Otherwise, the virtual machines will not have access to the network.

Related topics

Setting Up Kernel-Mode Debugging of a Virtual Machine Manually using a Virtual COM Port

Setting Up a Network Connection Manually

Setting Up Local Kernel Debugging of a Single Computer Manually

12/5/2018 • 2 minutes to read • Edit Online

Debugging Tools for Windows supports *local kernel debugging*. This is kernel-mode debugging on a single computer. In other words, the debugger runs on the same computer that is being debugged. With local debugging you can examine state, but not break into kernel mode processes that would cause the OS to stop running.

The *local* bcdedit option is available in Windows 8.0 and Windows Server 2012 and later.

Setting Up Local Kernel-Mode Debugging

> **IMPORTANT**
> Before using bcdedit to change boot information you may need to temporarily suspend Windows security features such as BitLocker and Secure Boot on the test PC. You can re-enable Secure Boot once you're done debugging and you've disabled kernel debugging on the local computer.

1. Open a Command Prompt window as Administrator. Enter **bcdedit /debug on**
2. If the computer is not already configured as the target of a debug transport, enter **bcdedit /dbgsettings local**
3. Reboot the computer.

Starting the Debugging Session

Using WinDbg

Open WinDbg as Administrator. On the **File** menu, choose **Kernel Debug**. In the Kernel Debugging dialog box, open the **Local** tab. Click **OK**.

You can also start a session with WinDbg by opening a Command Prompt window as Administrator and entering the following command:

windbg -kl

Using KD

Open a Command Prompt window as Administrator, and enter the following command:

kd -kl

Related topics

Local Kernel-Mode Debugging

Setting Up Kernel-Mode Debugging Manually

Supported Ethernet NICs for Network Kernel Debugging in Windows 10

12/7/2018 • 4 minutes to read • Edit Online

You can do kernel debugging over an Ethernet network cable when the target computer is running Windows. The target computer must have a supported network interface card (NIC) or network adapter.

During kernel debugging, the computer that runs the debugger is called the *host computer*, and the computer being debugged is called the *target computer*. To do kernel debugging over a network cable, the target computer must have a supported network adapter. When the target computer is running Windows, the network adapters listed here are supported for kernel debugging.

Version Information

The list of supported adapters is for the following versions of Windows

- Windows 10, version 1809
- Windows Server 2019

Adapter Support for Previous Releases of Windows

To determine which set of NICs is supported for any particular release of Windows, examine the `VerifiedNicList.xml` file that is in the debuggers directory installed by the WDK that shipped with that particular release of Windows. For 64 bit Windows, by default, it will be installed in this directory:

```
C:\Program Files (x86)\Windows Kits\10\Debuggers\x64\VerifiedNicList.xml
```

Checking the VerifiedNicList.xml that ships in the WDK for a particular release, is required because additional hardware support is added to new releases of Windows that is not present in previous releases. So you must check the VerifiedNicLIst.xml file for that particular release.

Finding the vendor ID and device ID

First find the vendor ID and device ID of the network adapter on your target computer.

- On the target computer, open Device Manager (enter **devmgmt** in a Command Prompt window).
- In Device Manager, locate the network adapter that you want to use for debugging.
- Right click the network adapter node, and choose **Properties**.
- In the **Details** tab, under **Property**, select **Hardware Ids**.

The vendor and device IDs are shown as VEN_*VendorID* and DEV_*DeviceID*. For example, if you see PCI\VEN_8086&DEV_104B, the vendor ID is 8086, and the device ID is 104B.

Vendor ID 8086, Intel Corporation

For vendor ID 8086, these device IDs are supported:

0001 0008 000C 000D 0438 043A 043C 0440 0470 1000 1001 1004 1008 1009 100C 100D 100E 100F 1010 1011 1012 1013 1014 1015 1016 1017 1018 1019 101A 101D 101E 1026 1027 1028 1049 104A 104B 104C 104D 105E 105F 1060 1071 1075 1076 1077 1078 1079 107A 107B 107C 107D 107E 107F 108A 108B 108C 1096 1098 1099 109A 10A4 10A5 10A7 10A9 10B5 10B9 10BA 10BB 10BC 10BD 10BF 10C0 10C2 10C3 10C4 10C5 10C6 10C7 10C8 10C9 10CB 10CC 10CD 10CE 10D3 10D5 10D6 10D9 10DA 10DB 10DD 10DE

10DF 10E1 10E5 10E6 10E7 10E8 10EA 10EB 10EC 10EF 10F0 10F1 10F4 10F5 10F6 10F7 10F8 10F9 10FB 10FC 11A9 1501 1502 1503 1507 150A 150B 150C 150D 150E 150F 1510 1511 1514 1516 1517 1518 151C 1521 1522 1523 1524 1525 1526 1527 1528 1529 152A 1533 1534 1535 1536 1537 1538 1539 153A 153B 1546 154A 154D 1557 1558 1559 155A 1560 1563 156F 1570 157B 157C 15A0 15A1 15A2 15A3 15AA 15AB 15AC 15AD 15AE 15B7 15B8 17D0 1F40 1F41 1F45 1F63 1F72 211B 2159 294C 8976

Vendor ID 10EC, Realtek Semiconductor Corp.

For vendor ID 10EC, these device IDs are supported:

8136 8137 8166 8167 8168 8169

Vendor ID 14E4, Broadcom

For vendor ID 14E4, these device IDs are supported:

1600 1601 1639 163A 163B 163C 163D 163E 1641 1642 1643 1644 1645 1646 1647 1648 164A 164C 164D 164E 164F 1650 1653 1654 1655 1656 1657 1659 165A 165B 165C 165D 165E 165F 1662 1663 1665 1668 1669 166A 166B 166D 166E 1672 1673 1674 1676 1677 1678 1679 167A 167B 167C 167D 167F 168A 168D 168E 1680 1681 1682 1683 1684 1686 1687 1688 1690 1691 1692 1693 1694 1696 1698 1699 169A 169B 169D 16A0 16A1 16A2 16A4 16A5 16A6 16A7 16A8 16AA 16AC 16AE 16B0 16B1 16B2 16B3 16B4 16B5 16B6 16B7 16C6 16C7 16DD 16F7 16FD 16FE 16FF 170D 170E 170F

Vendor ID 1969, Atheros Communications

For vendor ID 1969, these device IDs are supported:

1062 1063 1073 1083 1090 1091 10A0 10A1 10B0 10B1 10C0 10C1 10D0 10D1 10E0 10E1 10F0 10F1 2060 2062 E091 E0A1 E0B1 E0C1 E0D1 E0E1 E0F1

Vendor ID 19A2, ServerEngines (Emulex)

For vendor ID 19A2, these device IDs are supported:

0211 0215 0221 0700 0710

Vendor ID 10DF, Emulex Corporation

For vendor ID 10DF, these device IDs are supported:

0720 E220

Vendor ID 15B3, Mellanox Technology

For vendor ID 15B3, these device IDs are supported:

1000 1001 1002 1003 1004 1005 1006 1007 1008 1009 100A 100B 100C 100D 100E 100F 1010 1013 1015 1017 1019 101B 101D 101F 1021 1023 1025 1027 1029 102B 102F 6340 6341 634A 634B 6354 6368 6369 6372 6732 6733 673C 673D 6746 6750 6751 675A 6764 6765 676E 6778

Vendor ID 1137, Cisco Systems Inc

For vendor ID 1137, these device IDs are supported:

0043

Related topics

Setting Up Kernel-Mode Debugging over a Network Cable in Visual Studio

Setting Up KDNET Network Kernel Debugging Automatically

Setting Up KDNET Network Kernel Debugging Manually

Supported Ethernet NICs for Network Kernel Debugging in Windows 8

Supported Ethernet NICs for Network Kernel Debugging in Windows 8.1

Supported Ethernet NICs for Network Kernel Debugging in Windows 8.1

12/5/2018 • 3 minutes to read • Edit Online

You can do kernel debugging over an Ethernet network cable when the target computer is running Windows 8.1. The target computer must have a supported network interface card (NIC) or network adapter.

During kernel debugging, the computer that runs the debugger is called the *host computer*, and the computer being debugged is called the *target computer*. To do kernel debugging over a network cable, the target computer must have a supported network adapter. When the target computer is running Windows 8.1, the network adapters listed here are supported for kernel debugging.

Note Support for kernel debugging over selected 10 gigabit network adapters is a new feature in Windows 8.1. Debugging over 10 gigabit network adapters is not supported in Windows 8. For a list of network adapters supported by Windows 8 for kernel debugging, see Supported Ethernet NICs for Network Kernel Debugging in Windows 8.

System Requirements

Kernel debugging through Ethernet NICs requires certain low-level platform support. Windows requires that these NICs be attached via PCI/PCIe for this debugging solution. In most cases, simply plugging in one of these supported NICs will allow a robust kernel debugging experience. However, there may be cases where BIOS configuration details hinder the Windows debug path. The following platform requirement should be considered:

- System firmware should discover and configure the NIC device such that its resources do not conflict with any other devices that have been BIOS-configured.

Finding the vendor ID and device ID

First find the vendor ID and device ID of the network adapter on your target computer.

- On the target computer, open Device Manager (enter **devmgmt** in a Command Prompt window).
- In Device Manager, locate the network adapter that you want to use for debugging.
- Right click the network adapter node, and choose **Properties**.
- In the **Details** tab, under **Property**, select **Hardware Ids**.

The vendor and device IDs are shown as VEN_*VendorID* and DEV_*DeviceID*. For example, if you see PCI\VEN_8086&DEV_104B, the vendor ID is 8086, and the device ID is 104B.

Vendor ID 8086, Intel Corporation

For vendor ID 8086, these device IDs are supported:

0438 043A 043C 0440 1000 1001 1004 1008 1009 100C 100D 100E 100F 1010 1011 1012 1013 1014 1015 1016 1017 1018 1019 101A 101D 101E 1026 1027 1028 1049 104A 104B 104C 104D 105E 105F 1060 1075 1076 1077 1078 1079 107A 107B 107C 107D 107E 107F 108A 108B 108C 1096 1098 1099 109A 10A4 10A5 10A7 10A9 10B5 10B9 10BA 10BB 10BC 10BD 10BF 10C0 10C2 10C3 10C4 10C5 10C6 10C7 10C8 10C9 10CB 10CC 10CD 10CE 10D3 10D5 10D6 10D9 10DA 10DB 10DD 10DE 10DF 10E1 10E5 10E6 10E7 10E8 10EA 10EB 10EC 10EF 10F0 10F1 10F4 10F5 10F6 10F7 10F8 10F9 10FA 10FB 10FC 1501 1502 1503 1507

150A 150B 150C 150D 150E 150F 1510 1511 1514 1516 1517 1518 151C 1521 1522 1523 1524 1525 1526 1527 1528 1529 152A 1533 1534 1535 1536 1537 1538 1539 153A 153B 1546 154A 154D 1557 1558 1559 155A 1560 157B 157C 1F40 1F41 1F45 294C

Vendor ID 10EC, Realtek Semiconductor Corp.

For vendor ID 10EC, these device IDs are supported:

8136 8137 8167 8168 8169

Vendor ID 14E4, Broadcom

For vendor ID 14E4, these device IDs are supported:

1600 1601 1639 163A 163B 163C 1644 1645 1646 1647 1648 164A 164C 164D 1653 1654 1655 1656 1657 1658 1659 165A 165B 165C 165D 165E 165F 1668 1669 166A 166B 166D 166E 1672 1673 1674 1676 1677 1678 1679 167A 167B 167C 167D 167F 1680 1681 1684 1688 1690 1691 1692 1693 1694 1696 1698 1699 169A 169B 169D 16A0 16A6 16A7 16A8 16AA 16AC 16B0 16B1 16B2 16B4 16B5 16B6 16C6 16C7 16DD 16F7 16FD 16FE 16FF 170D 170E 170F

Vendor ID 1969, Atheros Communications

For vendor ID 1969, these device IDs are supported:

1062 1063 1073 1083 1090 1091 10A0 10A1 10B0 10B1 10C0 10C1 10D0 10D1 10E0 10E1 10F0 10F1 2060 2062 E091 E0A1 E0B1 E0C1 E0D1 E0E1 E0F1

Vendor ID 19A2, ServerEngines (Emulex)

For vendor ID 19A2, these device IDs are supported:

0211 0215 0221 0700 0710

Vendor ID 10DF, Emulex Corporation

For vendor ID 10DF, these device IDs are supported:

0720 E220

Vendor ID 15B3, Mellanox Technology

For vendor ID 15B3, these device IDs are supported:

1000 1001 1002 1003 1004 1005 1006 1007 1008 1009 100A 100B 100C 100D 100E 100F 1010 6340 6341 634A 634B 6354 6368 6369 6372 6732 6733 673C 673D 6746 6750 6751 675A 6764 6765 676E 6778

Related topics

Setting Up KDNET Network Kernel Debugging Automatically

Supported Ethernet NICs for Network Kernel Debugging in Windows 8

Supported Ethernet NICs for Network Kernel Debugging in Windows 8

12/5/2018 • 3 minutes to read • Edit Online

You can do kernel debugging over an Ethernet network cable when the target computer is running Windows 8. The target computer must have a supported network interface card (NIC) or network adapter.

During kernel debugging, the computer that runs the debugger is called the *host computer*, and the computer being debugged is called the *target computer*. To do kernel debugging over a network cable, the target computer must have a supported network adapter. When the target computer is running Windows 8, the network adapters listed here are supported for kernel debugging.

Note For a list of network adapters supported by Windows 8.1 for kernel debugging, see Supported Ethernet NICs for Network Kernel Debugging in Windows 8.1.

System Requirements

Kernel debugging through Ethernet NICs requires certain low-level platform support. Windows requires that these NICs be attached via PCI/PCIe for this debugging solution. In most cases, simply plugging in one of these supported NICs will allow a robust kernel debugging experience. However, there may be cases where BIOS configuration details hinder the Windows debug path. The following set of platform requirements should be considered:

- System firmware should discover and configure the NIC device such that its resources do not conflict with any other devices that have been BIOS-configured.
- System firmware should place the NIC's resources under address windows that are not marked prefetchable.

Finding the vendor ID and device ID

First find the vendor ID and device ID of the network adapter on your target computer.

- On the target computer, open Device Manager (enter **devmgmt** in a Command Prompt window).
- In Device Manager, locate the network adapter that you want to use for debugging.
- Right click the network adapter node, and choose **Properties**.
- In the **Details** tab, under **Property**, select **Hardware Ids**.

The vendor and device IDs are shown as VEN_*VendorID* and DEV_*DeviceID*. For example, if you see PCI\VEN_8086&DEV_104B, the vendor ID is 8086, and the device ID is 104B.

Vendor ID 8086, Intel Corporation

For vendor ID 8086, these device IDs are supported:

1000 1001 1004 1008 1009 100C 100D 100E 100F 1010 1011 1012 1013 1014 1015 1016 1017 1018 1019 101A 101D 101E 1026 1027 1028 1049 104A 104B 104C 104D 105E 105F 1060 1075 1076 1077 1078 1079 107A 107B 107C 107D 107E 107F 108A 108B 108C 1096 1098 1099 109A 10A4 10A5 10A7 10A9 10B5 10B9 10BA 10BB 10BC 10BD 10BF 10C9 10CB 10CC 10CD 10CE 10D3 10D5 10D6 10D9 10DA 10E5 10E6 10E7 10E8 10EA 10EB 10EF 10F0 10F5 10F6 1501 1502 1503 150A 150C 150D 150E 150F 1510 1511 1516 1518 1521 1522 1523 1524 1526 294C

Vendor ID 10EC, Realtek Semiconductor Corp.

For vendor ID 10EC, these device IDs are supported:

8136 8137 8167 8168 8169

Vendor ID 14E4, Broadcom

For vendor ID 14E4, these device IDs are supported:

1600 1601 1639 163A 163B 163C 1644 1645 1646 1647 1648 164A 164C 164D 1653 1654 1655 1656 1657 1658 1659 165A 165B 165C 165D 165E 165F 1668 1669 166A 166B 166D 166E 1672 1673 1674 1676 1677 1678 1679 167A 167B 167C 167D 167F 1680 1681 1684 1688 1690 1691 1692 1693 1694 1696 1698 1699 169A 169B 169D 16A0 16A6 16A7 16A8 16AA 16AC 16B0 16B1 16B2 16B4 16B5 16B6 16C6 16C7 16DD 16F7 16FD 16FE 16FF 170D 170E 170F

Vendor ID 1969, Atheros Communications

Support for Atheros network adapters is provided by a separate module that is available from Qualcomm. These device IDs are supported.

1062 1063 1073 1083 1090 1091 10A0 10A1 10B0 10B1 10C0 10C1 10D0 10D1 10E0 10E1 10F0 10F1 2060 2062 E091 E0A1 E0B1 E0C1 E0D1 E0E1 E0F1

Related topics

Setting Up KDNET Network Kernel Debugging Automatically

Supported Ethernet NICs for Network Kernel Debugging in Windows 8.1

Supported Ethernet NICs for Network Kernel Debugging in Windows 10

Configuring tools.ini

12/5/2018 • 3 minutes to read • Edit Online

The file tools.ini contains information to initialize the command-line debuggers. On startup, the debugger searches for the appropriate section header in the tools.ini file and extracts initialization information from the entries under the header. Each command-line debugger has its own section header - [CDB], [NTSD], and [KD]. The environment variable INIT must point to the directory containing the tools.ini file.

WinDbg does not use the tools.ini file. Instead, WinDbg saves initialization settings in workspaces.

The tools.ini entries are shown in the following table.

Keywords must be separated from the values by white space or a colon. Keywords are not case-sensitive.

For **TRUE** or **FALSE** values, "FALSE" is the only false value. Anything else is **TRUE**.

ENTRY	DESCRIPTION
$u0: *value* ... **$u9:** *value*	Assign values to fixed-name aliases. You can specify numeric values *n* or *0xn* or any other string. See Using Aliases for details. No command-line equivalent.
DebugChildren: *flag*	**TRUE** or **FALSE**. If **TRUE**, CDB debugs the specified application as well as any child processes that it might spawn. Command-line equivalent is **-o**.
DebugOutput: *flag*	**TRUE** or **FALSE**. If **TRUE**, CDB sends output and receives input through a terminal. If **FALSE**, output goes to the user screen. The command-line option **-d** is similar but not identical.
IniFile: *file*	Specifies the name of the script file that CDB or KD takes commands from at startup. The default is the ntsd.ini file in the current directory. Command-line equivalent is **-cf**. For details, see Using Script Files.
LazyLoad: *flag*	**TRUE** or **FALSE**. If **TRUE**, CDB performs lazy symbol loading; that is, symbols are not loaded until required. Command-line equivalent is **-s**. For details, and other methods of setting this option, see Deferred Symbol Loading.
SetDll: *filename*	Set extension DLL. The .dll filename extension should be omitted. Default is userexts.dll. Command-line equivalent is **-a**. For details, and other methods of setting this default, see Loading Debugger Extension DLLs.

ENTRY	DESCRIPTION
StopFirst: *flag*	**TRUE** or **FALSE**. If **true**, CDB stops on the breakpoint at the end of the image-loading process. Command-line equivalent is **-g**.
StopOnProcessExit: *flag*	**TRUE** or **FALSE**. If **TRUE**, CDB stops when it receives a process termination notification. Command-line equivalent is **-G**.
sxd: *event* **sxe:** *event*	Sets the debugger response and the handling status for the specified exception or event. Exceptions and events may be specified in the following ways: *****: Default exception *n*: Exception *n* (decimal) *0xn*: Exception *0xn* (hexadecimal) (other): Event code See Controlling Exceptions and Events for details of this process and for other methods of controlling these settings.
VerboseOutput: *flag*	**TRUE** or **FALSE**. If **TRUE**, CDB will display detailed information about symbol handling, event notification, and other run-time occurrences. Command-line equivalent is **-v**.
lines: *flag*	**TRUE** or **FALSE**. The lines flag enables or disables support for source-line information.
srcopt: *options*	Sets the source line options that control source display and program stepping options. For more information see **l+, l- (Set Source Options)**.
srcpath: *directory*	Sets the source file search path. For more information see **.srcpath, .lsrcpath (Set Source Path)**.
enable_unicode: *flag*	**TRUE** or **FALSE**. The enable_unicode flag specifies whether the debugger displays USHORT pointers and arrays as Unicode strings.
force_radix_output: *flag*	**TRUE** or **FALSE**. The force_radix_output flag specifies whether integers are displayed in decimal format or in the default radix.
col_mode: *flag*	**TRUE** or **FALSE**. The col_mode flag controls the color mode setting. When color mode is enabled the debugger can produce colored output. By default, most colors are not set and instead default to the current console colors.

ENTRY	DESCRIPTION
col: *name colspec*	The *name* indicates the element that you are coloring. The *colspec* is a three-letter RGB indicator of the form [rR-][gG-][bB-]. A lower-case letter indicates darker, an upper-case letter indicates brighter and a dash indicates no color component contribution. Due to console color limitations, bright is not actually per-component, but applies to all components if any request bright. In other words, rgB is the same as RGB. For this reason, it is recommended that all caps be used if any caps are going to be used. Example usage: col: emphfg R--

A sample [NTSD] section in the tools.ini file follows:

```
[NTSD]
sxe: 3c
sxe: cc
$u0: VeryLongName
VerboseOutput:true
```

Using KDbgCtrl

6/25/2019 • 5 minutes to read • Edit Online

The KDbgCtrl (Kernel Debugging Control, kdbgctrl.exe) tool can be used to control the kernel debugging connection from the target computer.

To use this tool, your target computer must be running Windows Server 2003 or a later version of Windows.

KDbgCtrl can control five different settings: Full Kernel Debugging, Automatic Kernel Debugging, User-Mode Error Handling, Blocking of Kernel Debugging, and the size of the DbgPrint buffer.

To use KDbgCtrl, you must have already enabled kernel debugging in the boot settings of the target computer before the last boot. KDbgCtrl cannot be used to enable kernel debugging if this was not done. See Boot Parameters to Enable Debugging for details on these boot settings.

Full Kernel Debugging

When Full Kernel Debugging is enabled, a kernel debugger running on the host computer can break into the target computer. The target computer will break into the kernel debugger if a kernel-mode exception is hit. Messages from the target to the host, such as **DbgPrint** output, symbol load messages, and redirected user-mode debuggers, are also allowed.

If this setting is disabled, all signals from the host computer will be ignored by the target.

Full Kernel Debugging is enabled by default. To check the current setting value, use **kdbgctrl -c**. To disable this setting, use **kdbgctrl -d**. To enable this setting, use **kdbgctrl -e**.

If you wish to check the current setting and use it to control execution within a batch file, you can use the **kdbgctrl -cx** command. For details on this command, see **KDbgCtrl Command-Line Options**.

Automatic Kernel Debugging

If Full Kernel Debugging is enabled, the current setting for Automatic Kernel Debugging is immaterial -- all communication is permitted.

When Full Kernel Debugging is disabled and Automatic Kernel Debugging is enabled, only the target computer can initiate a debugging connection.

In this case, only a kernel-mode exception, breakpoint, or other kernel-mode event will cause a connection to be established. The connection will not be established for **DbgPrint** output, symbol load messages, redirected user-mode debugger input and output, or other similar messages -- these will be stored in the DbgPrint buffer instead of being sent to the kernel debugger.

If an exception or event causes the target to break into the kernel debugger, then Full Kernel Debugging will be automatically turned on, just as if you had executed **kdbgctrl -e**.

Automatic Kernel Debugging is disabled by default (although this is immaterial unless Full Kernel Debugging is disabled as well). To check the current setting value, use **kdbgctrl -ca**. To disable this setting, use **kdbgctrl -da**. To enable this setting, use **kdbgctrl -ea**.

User-Mode Error Handling

When User-Mode Error Handling is enabled, some user-mode events will cause the target computer to break into the kernel debugger.

Specifically, all **int 3** interrupts -- such as breakpoints inserted into the code by a debugger or calls to **DbgBreakPoint** -- will cause a break into the kernel debugger. However, standard exceptions -- such as access

violations and division by zero -- will usually not be sent to the kernel debugger.

If a user-mode debugger is already attached to the process, this debugger will capture all user-mode errors, and the kernel debugger will not be alterted. For the precedence ranking of the various user-mode error handlers, see Enabling Postmortem Debugging.

For User-Mode Error Handling to function, either Full Kernel Debugging or Automatic Kernel Debugging must be enabled as well.

User-Mode Error Handling is enabled by default. To check the current setting value, use **kdbgctrl -cu**. To disable this setting, use **kdbgctrl -du**. To enable this setting, use **kdbgctrl -eu**.

Blocking Kernel Debugging

In some cases you might want to set up the target computer for kernel debugging, but wait to enable kernel debugging until after the target computer is started. You can do that by blocking kernel debugging.

> **IMPORTANT**
> Before using BCDEdit to change boot information you may need to temporarily suspend Windows security features such as BitLocker and Secure Boot on the test PC. Re-enable these security features when testing is complete and appropriately manage the test PC, when the security features are disabled.

To block kernel debugging, set up the target computer by using commands similar to the following:

```
bcdedit /debug on
bcdedit /dbgsettings 1394 channel:32 /start DISABLE /noumex
```

When you restart the target computer, it will be prepared for kernel debugging, but kernel debugging and User-Mode Error Handling will be disabled. At that point, a host computer will not be able to attach to the target computer, bug checks will not be caught by the kernel debugger, and user-mode exceptions will not cause a break in to the kernel debugger.

When you are ready, you can enable kernel debugging (without restarting the target computer) by entering the following commands.

```
kdbgctrl -db
kdbgctrl -e
```

Later, you can disable kernel debugging by entering the following commands.

```
kdbgctrl -d
kdbgctrl -eb
```

You can use **kdbgctrl -cb** to check whether kernel debugging is blocked.

The DbgPrint Buffer Size

The DbgPrint buffer stores messages that the target computer has sent to the kernel debugger.

If Full Kernel Debugging is enabled, these messages will automatically appear in the kernel debugger. But if this option is disabled, these messages will be stored in the buffer. At a later point in time, you can enable kernel debugging, connect to a kernel debugger, and use the **!dbgprint** extension to see the contents of this buffer. For more information about this buffer, see The DbgPrint Buffer.

The default size of the DbgPrint buffer is 4 KB on a free build of Windows, and 32 KB on a checked build of Windows. To determine the current buffer size, use **kdbgctrl -cdb**. To change the buffer size, use **kdbgctrl -**

sdb*Size*, where *Size* specifies the new buffer size. For syntax details, see **KDbgCtrl Command-Line Options**.

Examples

To display all the current settings, use the following command:

```
kdbgctrl -c -ca -cu -cb -cdb
```

To restore the default settings, use the following command:

```
kdbgctrl -e -da -eu -db -sdb 0x1000
```

To lock out the host computer so that it only is contacted on exceptions, use the following command:

```
kdbgctrl -d -ea -eu
```

To disable all kernel debugging, use the following command:

```
kdbgctrl -d -da
```

If you are disabling all kernel debugging, you may also wish to increase the size of the DbgPrint buffer. This insures that all messages will be saved in case you need to see them later. If you have a megabyte of memory to spare, you might use the following command:

```
kdbgctrl -sdb 0x100000
```

Debug Universal Drivers - Step by Step Lab (Echo Kernel-Mode)

10/23/2019 • 40 minutes to read • Edit Online

This lab introduces the WinDbg kernel debugger. WinDbg is used to debug the echo kernel mode sample driver code.

Lab objectives

This lab includes exercises that introduce the debugging tools, teach common debugging commands, illustrate the use of break points, and show the use of the debugging extensions.

In this lab, a live kernel debug connection is used to explore the following:

- Use the Windows debugger commands
- Use standard commands (Call stacks, variables, threads, IRQL)
- Use advanced driver debugging commands (!commands)
- Use symbols
- Set breakpoints in live debugging
- View call stacks
- Display the Plug and Play device tree
- Work with thread and process context

Note When working with the Windows debugger, there are two types of debugging that can be performed - user or kernel mode debugging.

User mode - Applications and subsystems run on the computer in user mode. Processes that run in user mode do so within their own virtual address spaces. They are restricted from gaining direct access to many parts of the system, including system hardware, memory that was not allocated for their use, and other portions of the system that might compromise system integrity. Because processes that run in user mode are effectively isolated from the system and other user mode processes, they cannot interfere with these resources.

Kernel mode - Kernel mode is the processor access mode in which the operating system and privileged programs run. Kernel mode code has permission to access any part of the system, and is not restricted like user mode code. It can gain access to any part of any other process running in either user mode or kernel mode. Much of the core OS functionality and many hardware device drivers run in kernel mode.

This lab will focus on kernel mode debugging, as that is the method used to debug many device drivers.

This exercise covers debug commands that are frequently used during both user-mode and kernel-mode debugging. The exercise also covers debug extensions (sometimes called "!commands") that are used for kernel-mode debugging.

Lab setup

You will need the following hardware to be able to complete the lab.

- A laptop or desktop computer (host) running Windows 10
- A laptop or desktop computer (target) running Windows 10
- A network hub/router and network cables to connect the two PCs

- Access to the internet to download symbol files

You will need the following software to be able to complete the lab.

- Visual Studio 2017
- Windows Software Development Kit (SDK) for Windows 10
- Windows Driver Kit (WDK) for Windows 10
- The sample echo driver for Windows 10

The lab has the following eleven sections.

- Section 1: Connect to a kernel mode WinDbg session
- Section 2: Kernel mode debugging commands and techniques
- Section 3: Download and build the KMDF Universal Echo Driver
- Section 4: Install the KMDF Echo driver sample on the target system
- Section 5: Use WinDbg to display information about the driver
- Section 6: Display Plug and Play device tree information
- Section 7: Work with breakpoints and source code
- Section 8: View variables and call stacks
- Section 9: Display processes and threads
- Section 10: IRQL, Registers and Ending the WinDbg session
- Section 11: Windows debugging resources

Section 1: Connect to a kernel mode WinDbg session

In Section 1, you will configure network debugging on the host and target system.

The PCs in this lab need to be configured to use an Ethernet network connection for kernel debugging.

This lab uses two PCs. Windows debugger runs on the *host* system and the KMDF Echo driver runs on the *target* system.

Use a network hub/router and network cables to connect the two PCs.

To work with kernel mode applications and use WinDbg, we recommend that you use the KDNET over Ethernet transport. For information about how to use the Ethernet transport protocol, see Getting Started with WinDbg (Kernel-Mode). For more information about setting up the target computer, see Preparing a Computer for Manual Driver Deployment and Setting Up KDNET Network Kernel Debugging Automatically.

Configure kernel–mode debugging using ethernet

To enable kernel mode debugging on the target system, perform the following steps.

<- On the host system

1. Open a command prompt on the host system and type **ipconfig** to determine its IP address.

```
C:\>ipconfig
Windows IP Configuration
Ethernet adapter Ethernet:
   Connection-specific DNS Suffix  . :
   Link-local IPv6 Address . . . . . : fe80::c8b6:db13:d1e8:b13b%3
   Autoconfiguration IPv4 Address. . : 169.182.1.1
   Subnet Mask . . . . . . . . . . . : 255.255.0.0
   Default Gateway . . . . . . . . . :
```

2. Record the IP address of the host system: _____

-> On the target system

3. Open a command prompt on the target system and use the **ping** command to confirm network connectivity between the two systems. Use the actual IP address of the host system you recorded instead of 169.182.1.1 that is shown in the sample output.

```
C:\> ping 169.182.1.1

Pinging 169.182.1.1 with 32 bytes of data:
Reply from 169.182.1.1: bytes=32 time=1ms TTL=255
Reply from 169.182.1.1: bytes=32 time<1ms TTL=255
Reply from 169.182.1.1: bytes=32 time<1ms TTL=255
Reply from 169.182.1.1: bytes=32 time<1ms TTL=255

Ping statistics for 169.182.1.1:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 0ms, Maximum = 1ms, Average = 0ms
```

Enable kernel mode debugging on the target system by completing the following steps.

> **IMPORTANT**
> Before using BCDEdit to change boot information you may need to temporarily suspend Windows security features such as BitLocker and Secure Boot on the test PC. Re-enable these security features when testing is complete and appropriately manage the test PC, when the security features are disabled.

1. On the target computer, open a Command Prompt window as Administrator. Enter this command to enable debugging.

```
C:\> bcdedit /set {default} DEBUG YES
```

2. Type this command to enable test signing.

```
C:\> bcdedit /set TESTSIGNING ON
```

3. Type this command to set the IP address of the host system. Use the IP address of the host system that you recorded earlier, not the one shown.

```
C:\> bcdedit /dbgsettings net hostip:192.168.1.1 port:50000 key:1.2.3.4
```

Warning To increase the security of the connection and decrease the risk of the random client debugger connection requests, consider using an auto generated random key. For more information, see Setting Up KDNET

Network Kernel Debugging Automatically.

4. Type this command to confirm that the dbgsettings they are set properly.

```
C:\> bcdedit /dbgsettings
key                     1.2.3.4
debugtype               NET
hostip                  169.168.1.1
port                    50000
dhcp                    Yes
The operation completed successfully.
```

Note
Firewalls and debuggers

If you receive a pop-up message from the firewall, and you wish to use the debugger, check **all three** of the boxes.

<- On the host system

1. On the host computer, open a Command Prompt window as Administrator. We will use the x64 version of WinDbg.exe from the Windows Driver Kit (WDK) that was installed as part of the Windows kit installation. By default it is located here.

```
C:\> Cd C:\Program Files(x86)\Windows Kits\10\Debuggers\x64
```

> **NOTE**
> This labs assumes that both PCs are running a 64 bit version of Windows on both the target and host. If that is not the case, the best approach is to run the same "bitness" of tools on the host that the target is running. For example if the target is running 32 bit Windows, run a 32 version of the debugger on the host. For more information, see Choosing the 32-Bit or 64-Bit Debugging Tools.

2. Launch WinDbg with remote user debug using the following command. The value for the key and port match what we set earlier using BCDEdit on the target.

```
WinDbg -k net:port=50000,key=1.2.3.4
```

->On the target system

Reboot the target system.

<-On the host system

In a minute or two, debug output should be displayed on the host system.

```
Microsoft (R) Windows Debugger Version 10.0.17074.1002 AMD64
Copyright (c) Microsoft Corporation. All rights reserved.

Using NET for debugging
Opened WinSock 2.0
Waiting to reconnect...
Connected to target 169.182.1.1 on port 50005 on local IP 169.182.1.2
You can get the target MAC address by running .kdtargetmac command.
Connected to Windows 10 16299 x64 target at (Wed Feb 28 17:16:23.051 2018 (UTC - 8:00)), ptr64 TRUE
Kernel Debugger connection established.  (Initial Breakpoint requested)
Symbol search path is: srv*
Executable search path is:
Windows 10 Kernel Version 16299 MP (4 procs) Free x64
Product: WinNt, suite: TerminalServer SingleUserTS
Built by: 16299.15.amd64fre.rs3_release.170928-1534
Machine Name:
Kernel base = 0xfffff800`9540d000 PsLoadedModuleList = 0xfffff800`95774110
Debug session time: Wed Feb 28 17:16:23.816 2018 (UTC - 8:00)
System Uptime: 0 days 0:00:20.534
```

The Debugger Command window is the primary debugging information window in WinDbg. You can enter debugger commands and view the command output in this window.

The Debugger Command window is split into two panes. You type commands in the smaller pane (the command entry pane) at the bottom of the window and view the command output in the larger pane at the top of the window.

In the command entry pane, use the up arrow and down arrow keys to scroll through the command history. When a command appears, you can edit it or press **ENTER** to run the command.

Section 2: Kernel mode debugging commands and techniques

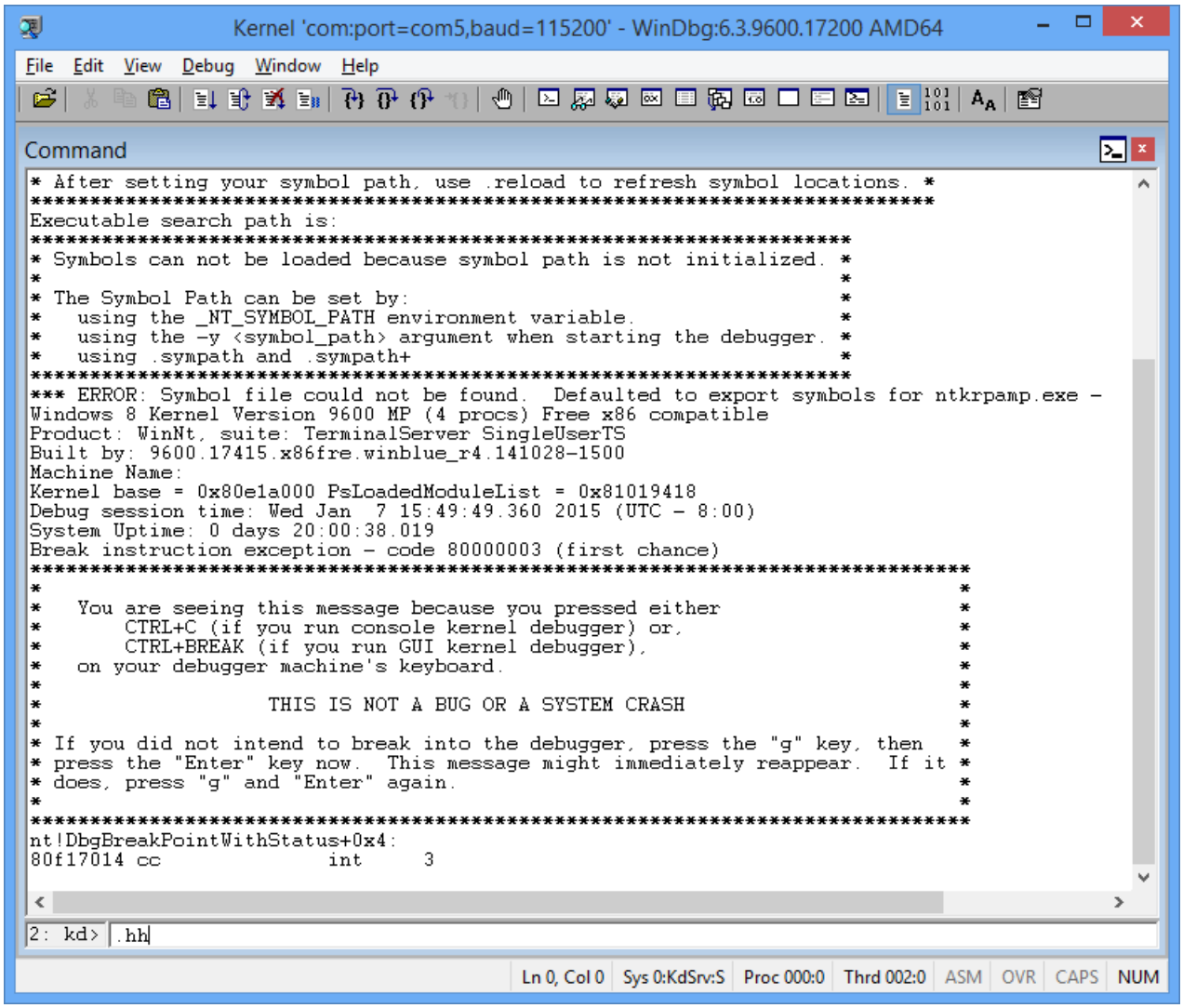
In Section 2, you will use debug commands to display information about the target system.

<- On the host system

Enable Debugger Markup Language (DML) with .prefer_dml

Some debug commands display text using Debugger Markup Language that you can click on to quickly gather more information.

1. Use Ctrl+Break (Scroll Lock) in WinDBg to break into the code running on the target system. It may take a bit of time for the target system to respond.



2. Type the following command to enable DML in the Debugger Command window.

```
0: kd> .prefer_dml 1
DML versions of commands on by default
```

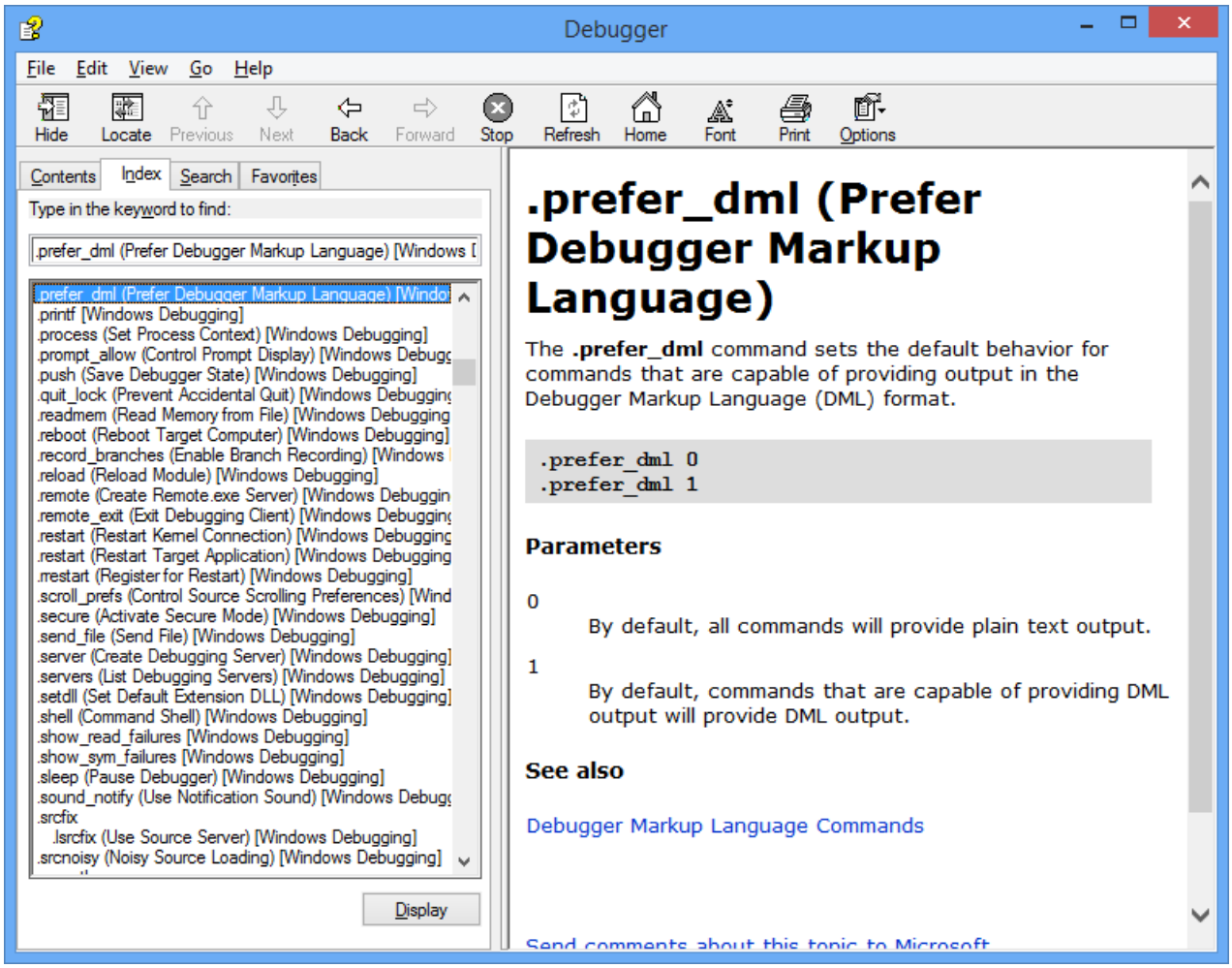
Use .hh to get help

You can access reference command help using the **.hh** command.

3. Type the following command to view the command reference help for **.prefer_dml**.

```
0: kd> .hh .prefer_dml
```

The Debugger help file will display help for the **.prefer_dml** command.



Display the version of Windows on the target system

5. Display detailed version information on the target system by typing the **vertarget (Show Target Computer Version)** command in the WinDbg window.

```
0: kd> vertarget
Windows 10 Kernel Version 9926 MP (4 procs) Free x64
Product: WinNt, suite: TerminalServer SingleUserTS
Built by: 9926.0.amd64fre.fbl_awesome1501.150119-1648
Machine Name: ""
Kernel base = 0xfffff801`8d283000 PsLoadedModuleList = 0xfffff801`8d58aef0
Debug session time: Fri Feb 20 10:15:17.807 2015 (UTC - 8:00)
System Uptime: 0 days 01:31:58.931
```

List the loaded modules

6. You can verify that you are working with the right kernel mode process by displaying the loaded modules by typing the **lm (List Loaded Modules)** command in the WinDbg window.

```
0: Kd> lm
start             end                 module name
fffff801`09200000 fffff801`0925f000   volmgrx    (no symbols)
fffff801`09261000 fffff801`092de000   mcupdate_GenuineIntel   (no symbols)
fffff801`092de000 fffff801`092ec000   werkernel   (export symbols)       werkernel.sys
fffff801`092ec000 fffff801`0934d000   CLFS       (export symbols)       CLFS.SYS
fffff801`0934d000 fffff801`0936f000   tm         (export symbols)       tm.sys
fffff801`0936f000 fffff801`09384000   PSHED      (export symbols)       PSHED.dll
fffff801`09384000 fffff801`0938e000   BOOTVID    (export symbols)       BOOTVID.dll
fffff801`0938e000 fffff801`093f7000   spaceport   (no symbols)
fffff801`09400000 fffff801`094cf000   Wdf01000   (no symbols)
fffff801`094d9000 fffff801`09561000   CI         (export symbols)       CI.dll
...
```

Note Output that has been omitted is indicated with "... " in this lab.

7. To request detailed information about a specific module, use the v (verbose) option as shown.

```
0: Kd> lm v m tcpip
Browse full module list
start             end                 module name
fffff801`09eeb000 fffff801`0a157000   tcpip      (no symbols)
    Loaded symbol image file: tcpip.sys
    Image path: \SystemRoot\System32\drivers\tcpip.sys
    Image name: tcpip.sys
    Browse all global symbols  functions  data
    Timestamp:        Sun Nov 09 18:59:03 2014 (546029F7)
    CheckSum:         00263DB1
    ImageSize:        0026C000
    Translations:     0000.04b0 0000.04e4 0409.04b0 0409.04e4

Unable to enumerate user-mode unloaded modules, Win32 error 0n30
```

8. Because we have yet to set the symbol path and loaded symbols, limited information is available in the debugger.

Section 3: Download and build the KMDF universal echo driver

In Section 3, you will download and build the KMDF universal echo driver.

Typically, you would be working with your own driver code when you use WinDbg. To become familiar with WinDbg operation, the KMDF Template "Echo" sample driver is used. With the source code available, it will also be easier to understand the information that is displayed in WinDbg. In addition, this sample is used to illustrate how you can single step through native kernel mode code. This technique can be very valuable for debugging complex kernel mode code issues.

To download and build the Echo sample audio driver, perform the following steps.

1. **Download and extract the KMDF Echo sample from GitHub**

   You can use a browser to view the echo sample in GitHub here:

   https://github.com/Microsoft/Windows-driver-samples/tree/97cf5197cf5b882b2c689d8dc2b555f2edf8f418/general/echo/kmdf

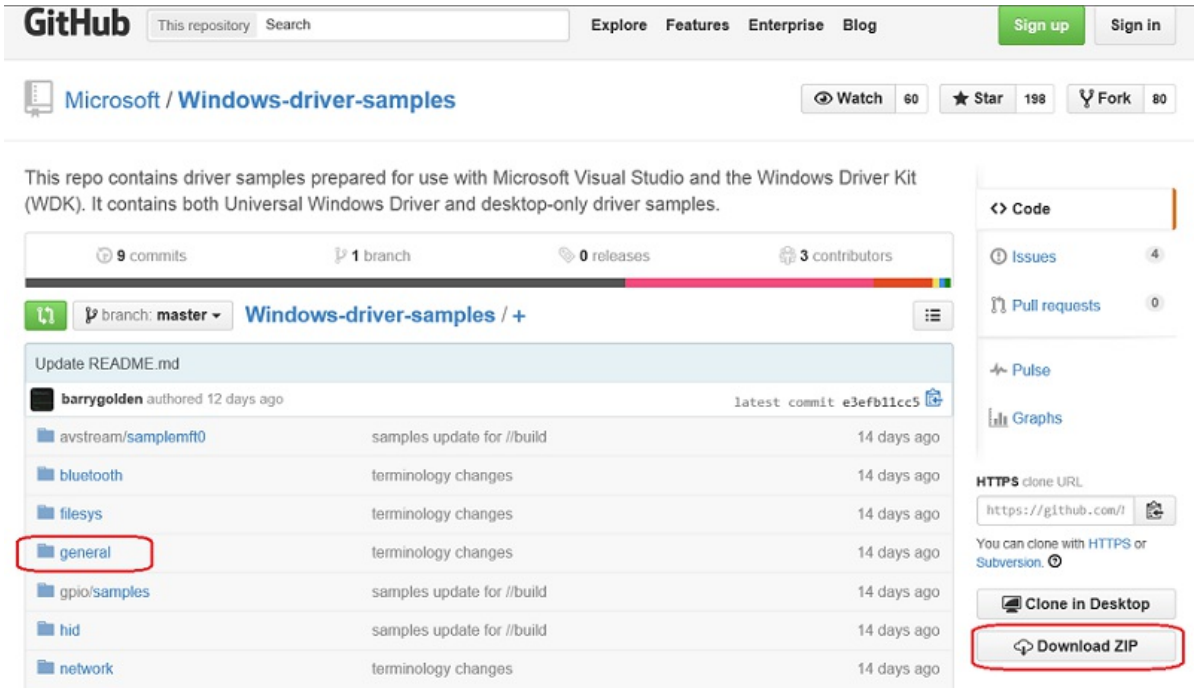
   You can read about the sample here:

   https://github.com/Microsoft/Windows-driver-samples/blob/97cf5197cf5b882b2c689d8dc2b555f2edf8f418/general/echo/kmdf/ReadMe.md

   You can browse all of the universal driver samples here:

https://github.com/Microsoft/Windows-driver-samples

The KMDF Echo sample is located in the general folder.



a. This lab, shows how to download the universal driver samples in one zip file.

https://github.com/Microsoft/Windows-driver-samples/archive/master.zip

b. Download the master.zip file to your local hard drive.

c. Right-click *Windows-driver-samples-master.zip*, and choose **Extract All**. Specify a new folder, or browse to an existing one that will store the extracted files. For example, you could specify *C:\DriverSamples* as the new folder into which the files are extracted.

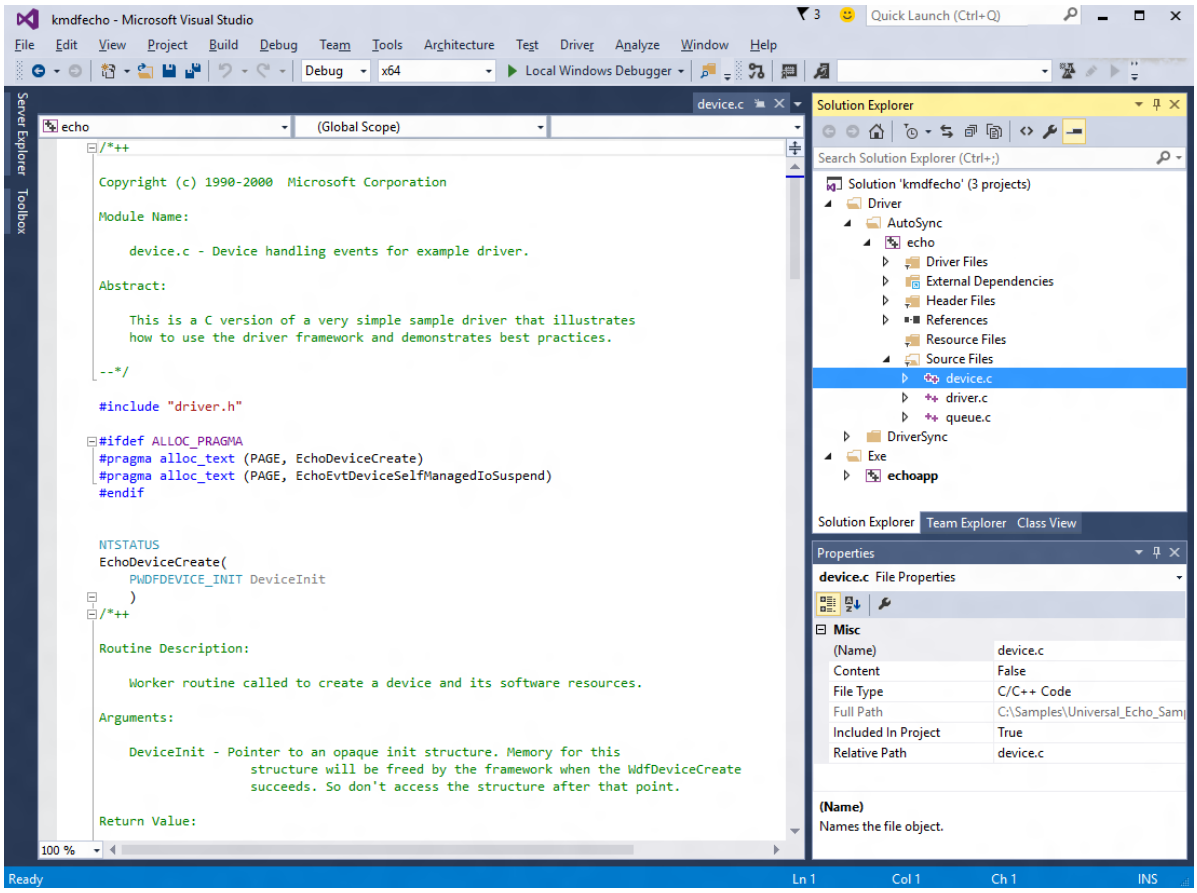
d. After the files are extracted, navigate to the following subfolder.

C:\DriverSamples\general\echo\kmdf

2. **Open the driver solution in Visual Studio**

   In Microsoft Visual Studio, click **File** > **Open** > **Project/Solution...** and navigate to the folder that contains the extracted files (for example, *C:\DriverSamples\general\echo\kmdf*). Double-click the *kmdfecho* solution file to open it.

   In Visual Studio, locate the Solution Explorer. (If this is not already open, choose **Solution Explorer** from the **View** menu.) In Solution Explorer, you can see one solution that has three projects.

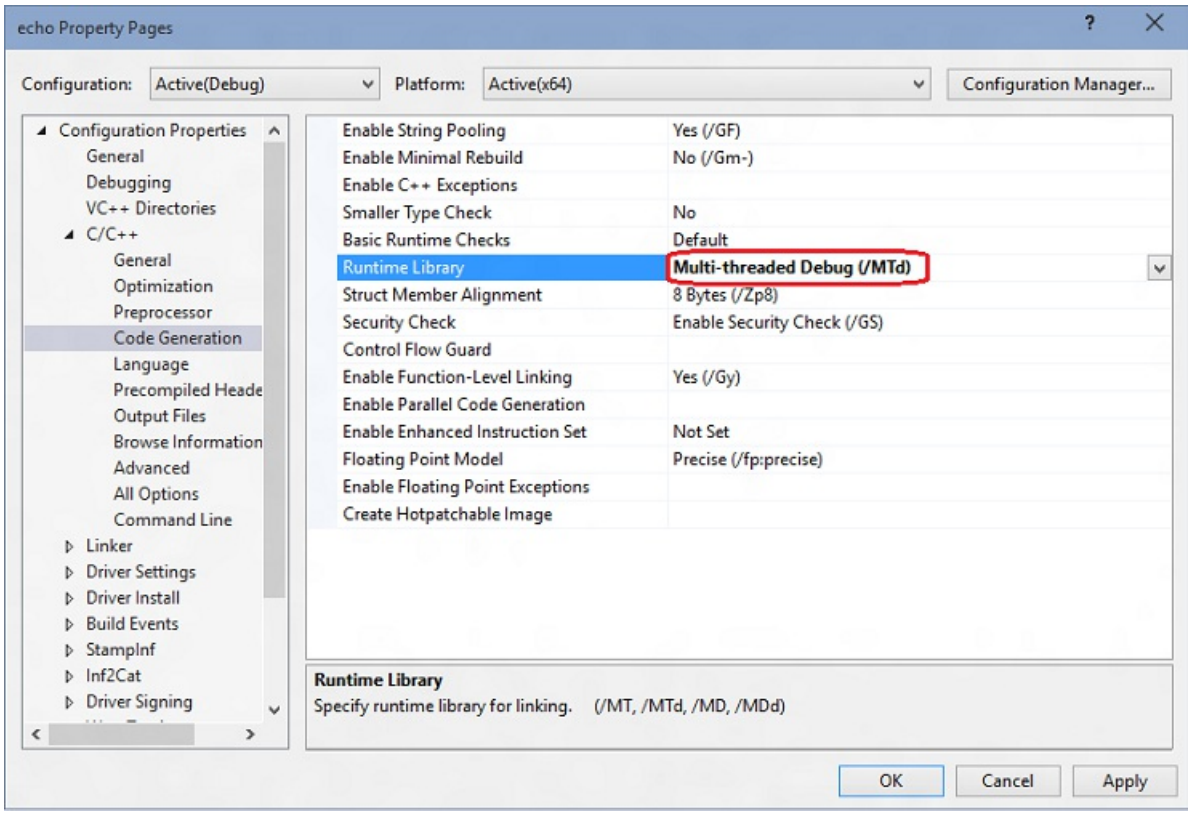


3. **Set the sample's configuration and platform**

   In Solution Explorer, right-click **Solution 'kmdfecho' (3 projects)**, and choose **Configuration Manager**. Make sure that the configuration and platform settings are the same for the three projects. By default, the configuration is set to "Win10 Debug", and the platform is set to "Win64" for all the projects. If you make any configuration and/or platform changes for one project, you must make the same changes for the remaining three projects.

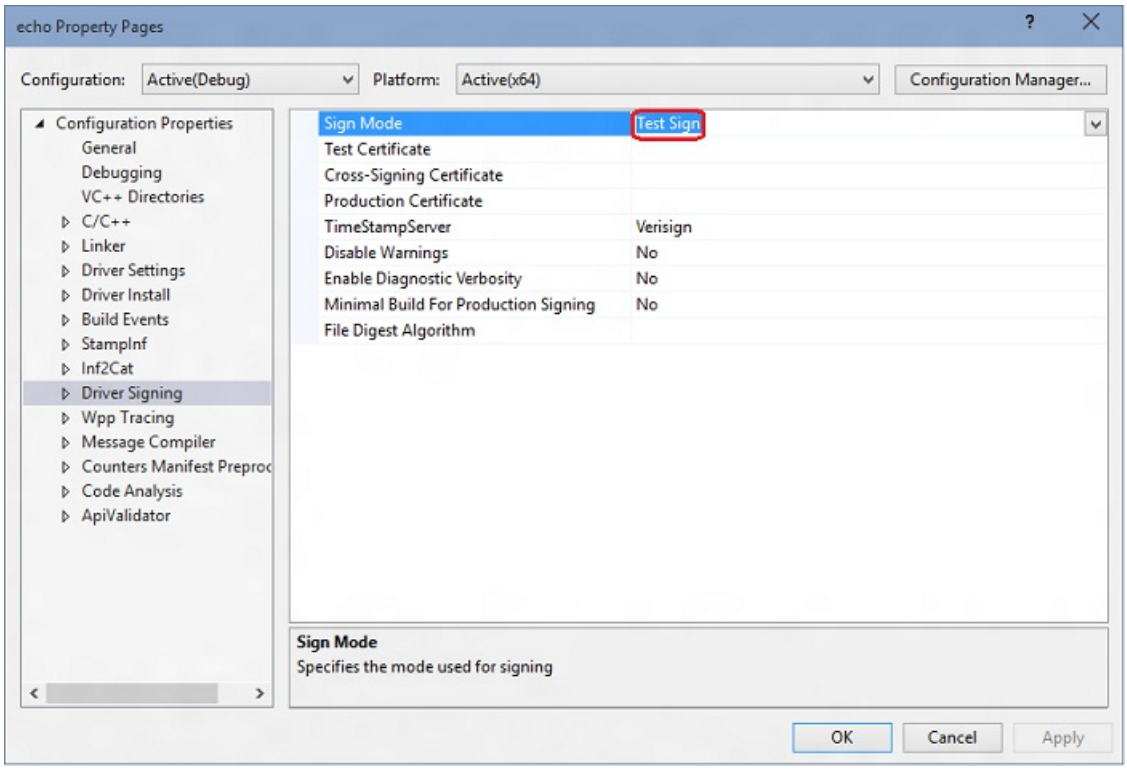
4. **Set the runtime library**

   Set the runtime library - Open the echo driver's property page and locate **C/C++** > **Code Generation**. Change Runtime Library from DLL version to non DLL version. Without this setting, you have to install the MSVC runtime to the target computer separately.



5. **Check driver signing**

   Also on the driver's properties make sure **Driver Signing** > **Sign Mode** is set to "Test Sign". This is required because Windows requires that drivers are signed.



6. **Build the sample using Visual Studio**

   In Visual Studio, click **Build** > **Build Solution**.

   If all goes well, the build windows should display a message indicating that the build for all three projects succeeded.

7. **Locate the built driver files**

In File Explorer, navigate to the folder that contains the extracted files for the sample. For example, you would navigate to *C:\DriverSamples\general\echo\kmdf*, if that's the folder you specified earlier. Within that folder, the location of the compiled driver files varies depending on the configuration and platform settings that you selected in the **Configuration Manager**. For example, if you left the default settings unchanged, then the compiled driver files will be saved to a folder named *\x64\Debug* for a 64 bit, debug build.

Navigate to the folder that contains the built files for the Autosync driver:

C:\DriverSamples\general\echo\kmdf\driver\AutoSync\x64\Debug.

The folder should contain these files:

FILE	DESCRIPTION
Echo.sys	The driver file.
Echo.inf	An information (INF) file that contains information needed to install the driver.

In addition, the echoapp.exe file was built and it should be located here:
C:\DriverSamples\general\echo\kmdf\exe\x64\Debug

FILE	DESCRIPTION
EchoApp.exe	A command prompt executable test file that communicates with the echo.sys driver.

8. Locate a USB thumb drive or set up a network share to copy the built driver files and the test EchoApp from the host to the target system.

In the next section, you will copy the code to the target system, and install and test the driver.

Section 4: Install the KMDF echo driver sample on the target system

In Section 4, you will use devcon to install the echo sample driver.

-> On the target system

The computer where you install the driver is called the *target computer* or the *test computer*. Typically, this is a separate computer from the computer on which you develop and build the driver package. The computer where you develop and build the driver is called the *host computer*.

The process of moving the driver package to the target computer and installing the driver is called *deploying* the driver. You can deploy the sample echo driver, automatically or manually.

Before you manually deploy a driver, you must prepare the target computer by turning on test signing. You also need to locate the DevCon tool in your WDK installation. After that you're ready to run the built driver sample.

To install the driver on the target system, perform the following steps.

Enable test signed drivers

Enable the ability to run test signed drivers:

a. Open Windows Settings.

b. In Update and Security, select **Recovery**.

c. Under Advanced startup, click **Restart Now**.

d. When the PC reboots, select **Startup options**. In Windows 10, select **Troubleshoot** > **Advanced options** > **Startup Settings** , then click Restart button.

e. Select Disable driver signature enforcement by pressing the **F7** key.

f. Reboot the target computer.

<- On the host system

Navigate to the Tools folder in your WDK installation and locate the DevCon tool. For example, look in the following folder:

C:\Program Files (x86)\Windows Kits\10\Tools\x64\devcon.exe Create a folder on the target for the built driver package (for example, *C:\EchoDriver*). Copy all the files from the built driver described earlier on the host computer and save them to the folder that you created on the target computer.

Locate the .cer certificate on the host system, it is in the same folder on the host computer in the folder that contains the built driver files. On the target computer, right-click the certificate file, and click **Install**, then follow the prompts to install the test certificate.

If you need more detailed instructions for setting up the target computer, see Preparing a Computer for Manual Driver Deployment.

-> On the target system

Install the driver

The following instructions show you how to install and test the sample driver. Here's the general syntax for the devcon tool that you will use to install the driver:

devcon install <INF file> <hardware ID>

The INF file required for installing this driver is *echo.inf*. The inf file contains the hardware ID for installing the *echo.sys*. For the echo sample the hardware ID is **root\ECHO**.

On the target computer, open a Command Prompt window as Administrator. Navigate to your driver package folder, and enter the following command:

devcon install echo.inf root\ECHO If you get an error message about *devcon* not being recognized, try adding the path to the *devcon* tool. For example, if you copied it to a folder called *C:\Tools*, then try using the following command:

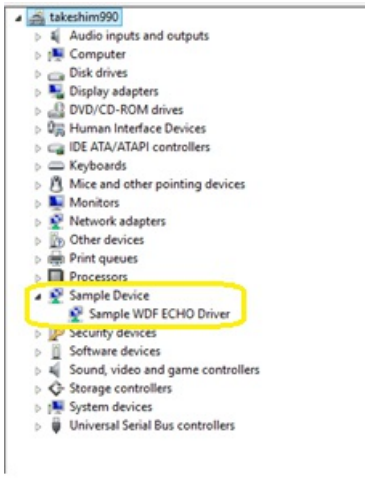
c:\tools\devcon install echo.inf root\ECHO A dialog box will appear indicating that the test driver is an unsigned driver. Click **Install this driver anyway** to proceed.

For more detailed instructions, see Configuring a Computer for Driver Deployment, Testing, and Debugging.

After successfully installing the sample driver, you're now ready to test it.

Examine the driver in Device Manager

On the target computer, in a Command Prompt window, enter **devmgmt** open Device Manager. In Device Manager, on the View menu, choose Devices by type. In the device tree, locate *Sample WDF Echo Driver* in the Sample Device node.



Test the driver

Type **echoapp** to start the test echo app to confirm that the driver is functional.

```
C:\Samples\KMDF_Echo_Sample> echoapp
DevicePath: \\?\root#sample#0005#{cdc35b6e-0be4-4936-bf5f-5537380a7c1a}
Opened device successfully
512 Pattern Bytes Written successfully
512 Pattern Bytes Read successfully
Pattern Verified successfully
30720 Pattern Bytes Written successfully
30720 Pattern Bytes Read successfully
Pattern Verified successfully
```

Section 5: Use WinDbg to display information about the driver

In Section 5, you will set the symbol path and use kernel debugger commands to display information about the KMDF echo sample driver.

View information about the driver by performing the following steps.

<-On the host system

1. If you closed the debugger, open it again using the following command in the administrator command prompt window.

   ```
   WinDbg -k net:port=50000,key=1.2.3.4
   ```

2. Use Ctrl+Break (Scroll Lock) to break into the code running on the target system.

Setting the symbol path

1. To set the symbols path to the Microsoft symbol server in the WinDbg environment, use the **.symfix** command.

```
0: kd> .symfix
```

2. To add your local symbol location to use your local symbols, add the path using **.sympath+** and then **.reload /f**.

```
0: kd> .sympath+ C:\DriverSamples\general\echo\kmdf
0: kd> .reload /f
```

Note The **.reload** command with the **/f** force option deletes all symbol information for the specified module and reloads the symbols. In some cases, this command also reloads or unloads the module itself.

Note You must load the proper symbols to use advanced functionality that WinDbg provides. If you do not have symbols properly configured, you will receive messages indicating that symbols are not available when you attempt to use functionality that is dependent on symbols.

```
0:000> dv
Unable to enumerate locals, HRESULT 0x80004005
Private symbols (symbols.pri) are required for locals.
Type “.hh dbgerr005” for details.
```

Note
Symbol servers

There are a number of approaches that can be used to work with symbols. In many situations, you can configure the PC to access symbols from a symbol server that Microsoft provides when they are needed. This walkthrough assumes that this approach will be used. If the symbols in your environment are in a different location, modify the steps to use that location. For additional information, see Symbol Stores and Symbol Servers.

Note
Understand source code symbol requirements

To perform source debugging, you must build a checked (debug) version of your binaries. The compiler will create symbol files (.pdb files). These symbol files will show the debugger how the binary instructions correspond to the source lines. The actual source files themselves must also be accessible to the debugger.

The symbol files do not contain the text of the source code. For debugging, it is best if the linker does not optimize your code. Source debugging and access to local variables are more difficult, and sometimes nearly impossible, if the code has been optimized. If you are having problems viewing local variables or source lines, set the following build options:

```
set COMPILE_DEBUG=1
set ENABLE_OPTIMIZER=0
```

1. Type the following in the command area of the debugger to display information about the echo driver :

```
0: kd> lm m echo* v
Browse full module list
start             end                 module name
fffff801`4ae80000 fffff801`4ae89000   ECHO       (private pdb symbols)
C:\Samples\KMDF_ECHO_SAMPLE\echo.pdb
    Loaded symbol image file: ECHO.sys
    Image path: \SystemRoot\system32\DRIVERS\ECHO.sys
    Image name: ECHO.sys
...
```

For information, see **lm**.

2. Because we set prefer_dml =1 earlier, some elements of the output are hot links that you can click on. Click on the *Browse all global symbols link* in the debug output to display information about items symbols that start with the letter "a".

```
0: kd> x /D Echo!a*
```

3. As it turns out, the echo sample doesn't contain any symbols that start with the letter "a", so to display information about all of the symbols associated with echo driver that start with Echo, type **x ECHO!Echo***.

```
0: kd> x ECHO!Echo*
fffff801`0bf95690 ECHO!EchoEvtIoQueueContextDestroy (void *)
fffff801`0bf95000 ECHO!EchoEvtDeviceSelfManagedIoStart (struct WDFDEVICE__ *)
fffff801`0bf95ac0 ECHO!EchoEvtTimerFunc (struct WDFTIMER__ *)
fffff801`0bf9b120 ECHO!EchoEvtDeviceSelfManagedIoSuspend (struct WDFDEVICE__ *)
...
```

For information, see **x (Examine Symbols)**.

4. The **!lmi** extension displays detailed information about a module. Type **!lmi echo**. Your output should be similar to the text shown below.

```
0: kd> !lmi echo
Loaded Module Info: [echo]
         Module: ECHO
   Base Address: fffff8010bf94000
     Image Name: ECHO.sys
…
```

5. Use the **!dh** extension to display header information as shown below.

```
0: kd> !dh echo

File Type: EXECUTABLE IMAGE
FILE HEADER VALUES
     14C machine (i386)
       6 number of sections
54AD8A42 time date stamp Wed Jan 07 11:34:26 2015
...
```

6. **Setting the debug mask**

   Type the following to change the default debug bit mask so that all debug messages from the target system will be displayed in the debugger.

```
0: kd> ed nt!Kd_DEFAULT_MASK  0xFFFFFFFF
```

   Some drivers will display additional information when the mask of 0xFFFFFFFF is used. Set the mask to 0x00000000 if you would like to reduce the amount of information that is displayed.

```
0: kd> ed nt!Kd_DEFAULT_MASK  0x00000000
```

   Use the dd command to display confirm the mask is set to display all of the debugger messages.

```
0: kd> dd nt!kd_DEFAULT_MASK
fffff802`bb4057c0  ffffffff 00000000 00000000 00000000
fffff802`bb4057d0  00000000 00000000 00000000 00000000
fffff802`bb4057e0  00000001 00000000 00000000 00000000
fffff802`bb4057f0  00000000 00000000 00000000 00000000
fffff802`bb405800  00000000 00000000 00000000 00000000
fffff802`bb405810  00000000 00000000 00000000 00000000
fffff802`bb405820  00000000 00000000 00000000 00000000
fffff802`bb405830  00000000 00000000 00000000 00000000
```

Section 6: Displaying Plug and Play device tree information

In Section 6, you will display information about the echo sample device driver and where it lives in the Plug and Play device tree.

Information about the device driver in the Plug and Play device tree can be useful for troubleshooting. For example, if a device driver is not resident in the device tree, there may an issue with the installation of the device driver.

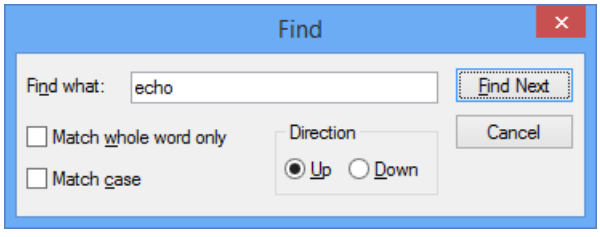
For more information about the device node debug extension, see **!devnode**.

<-On the host system

1. To see all the device nodes in the Plug and Play device tree, enter the **!devnode 0 1** command.

```
0: kd> !devnode 0 1
Dumping IopRootDeviceNode (= 0xffffe0005a3a8d30)
DevNode 0xffffe0005a3a8d30 for PDO 0xffffe0005a3a9e50
  InstancePath is "HTREE\ROOT\0"
  State = DeviceNodeStarted (0x308)
  Previous State = DeviceNodeEnumerateCompletion (0x30d)
  DevNode 0xffffe0005a3a3d30 for PDO 0xffffe0005a3a4e50
    InstancePath is "ROOT\volmgr\0000"
    ServiceName is "volmgr"
    State = DeviceNodeStarted (0x308)
    Previous State = DeviceNodeEnumerateCompletion (0x30d)
    DevNode 0xffffe0005a324560 for PDO 0xffffe0005bd95ca0…
…
```

2. Use Ctrl+F to search in the output that is generated to look for the name of the device driver, *echo*.



3. The echo device driver should be loaded. Use the **!devnode 0 1 echo** command to display Plug and Play information associated with our echo device driver as shown below.

```
0: Kd> !devnode 0 1 echo
Dumping IopRootDeviceNode (= 0xffffe0007b725d30)
DevNode 0xffffe0007b71a630 for PDO 0xffffe0007b71a960
  InstancePath is "ROOT\SAMPLE\0000"
  ServiceName is "ECHO"
  State = DeviceNodeStarted (0x308)
  Previous State = DeviceNodeEnumerateCompletion (0x30d)
…
```

4. The output displayed in the previous command includes the PDO associated with the running instance of our driver, in this example it is *0xffffe0007b71a960*. Enter the **!devobj**<*PDO address*> command to display Plug and Play information associated with the echo device driver. Use the PDO address that **!devnode** displays on your PC, not the one shown here.

```
0: kd> !devobj 0xffffe0007b71a960
Device object (ffffe0007b71a960) is for:
 0000000e \Driver\PnpManager DriverObject ffffe0007b727e60
Current Irp 00000000 RefCount 0 Type 00000004 Flags 00001040
Dacl ffffc102c9b36031 DevExt 00000000 DevObjExt ffffe0007b71aab0 DevNode ffffe0007b71a630
ExtensionFlags (0x00000800)  DOE_DEFAULT_SD_PRESENT
Characteristics (0x00000180)  FILE_AUTOGENERATED_DEVICE_NAME, FILE_DEVICE_SECURE_OPEN
AttachedDevice (Upper) ffffe000801fee20 \Driver\ECHO
Device queue is not busy.
```

5. The output displayed in the **!devnode 0 1** command includes the PDO address associated with the running instance of our driver, in this example it is *0xffffe0007b71a960*. Enter the **!devstack**<*PDO address*> command to display Plug and Play information associated with the device driver. Use the PDO address that **!devnode** displays on your PC, not the one shown below.

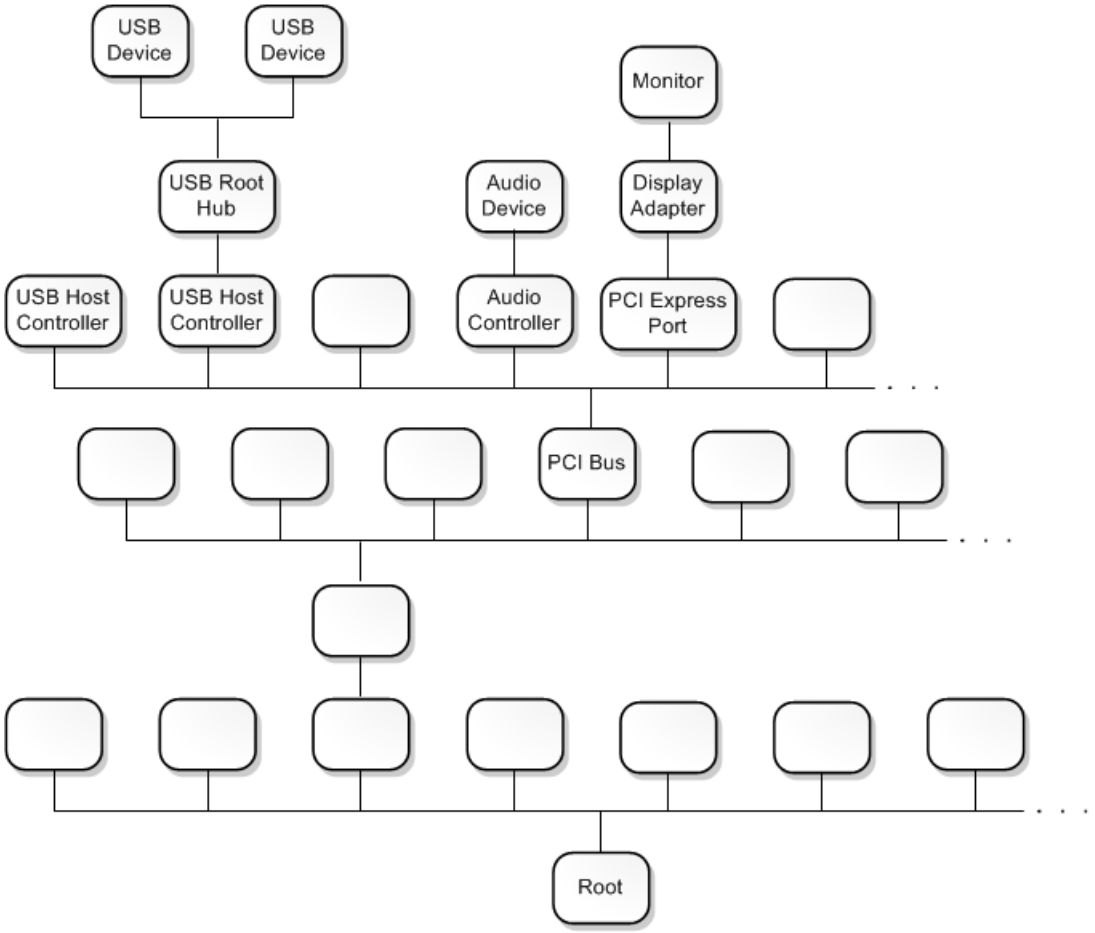
```
0: kd> !devstack 0xffffe0007b71a960
  !DevObj           !DrvObj            !DevExt           ObjectName
  ffffe000801fee20  \Driver\ECHO       ffffe0007f72eff0
> ffffe0007b71a960  \Driver\PnpManager 00000000  0000000e
!DevNode ffffe0007b71a630 :
  DeviceInst is "ROOT\SAMPLE\0000"
  ServiceName is "ECHO"
```

The output shows that we have a fairly simple device driver stack. The echo driver is a child of the PnPManager node. The PnPManager is a root node.

\Driver\ECHO

\Driver\PnpManager

This diagram shows a more complex device node tree.



Note For more information about more complex driver stacks, see Driver stacks and Device nodes and device stacks.

Section 7: Working with breakpoints and source code

In Section 7, you will set breakpoints and single step through kernel mode source code.

Note
Setting breakpoints using commands

To be able to step through code and check the values of variables in real time, we need to enable breakpoints and set a path to the source code.

Breakpoints are used to stop code execution at a particular line of code. You can then step forward in the code from that point, to debug that specific section of code.

To set a breakpoint using a debug command, use one of the following **b** commands.

bp	Sets a breakpoint that will be active until the module it is in is unloaded.
bu	Sets a breakpoint that is unresolved when the module is unloaded and re-enables when the module reloads.
bm	Sets a breakpoint for a symbol. This command will use bu or bp appropriately and allows wildcards * to be used to set breakpoints on every symbols that matches (like all methods in a class).

For more information, see Source Code Debugging in WinDbg in the debugging reference documentation.

<-On the host system

1. Use the WinDbg UI to confirm that **Debug** > **Source Mode** is enabled in the current WinDbg session.
2. Add your local code location to the source path by typing the following command.

```
.srcpath+ C:\DriverSamples\KMDF_Echo_Sample\driver\AutoSync
```

3. Add your local symbol location to the symbol path by typing the following command.

```
.sympath+ C:\DriverSamples\KMDF_Echo_Sample\driver\AutoSync
```

4. We will use the **x** command to examine the symbols associated with the echo driver to determine the function name to use for the breakpoint. We can use a wild card or Ctrl+F to locate the **DeviceAdd** function name.

```
0: kd> x ECHO!EchoEvt*
8b4c7490          ECHO!EchoEvtIoQueueContextDestroy (void *)
8b4c7000          ECHO!EchoEvtDeviceSelfManagedIoStart (struct WDFDEVICE__ *)
8b4c7820          ECHO!EchoEvtTimerFunc (struct WDFTIMER__ *)
8b4cb0e0          ECHO!EchoEvtDeviceSelfManagedIoSuspend (struct WDFDEVICE__ *)
8b4c75d0          ECHO!EchoEvtIoWrite (struct WDFQUEUE__ *, struct WDFREQUEST__ *, unsigned int)
8b4cb170          ECHO!EchoEvtDeviceAdd (struct WDFDRIVER__ *, struct
…
```

   The output above shows that **DeviceAdd** method for our echo driver is *ECHO!EchoEvtDeviceAdd*.

   Alternatively, we could review the source code to locate the desired function name for our breakpoint.

5. Set the breakpoint with the **bm** command using the name of the driver, followed by the function name (for example **AddDevice**) where you want to set the breakpoint, separated by an exclamation mark. We will use **AddDevice** to watch the driver being loaded.

```
0: kd> bm ECHO!EchoEvtDeviceAdd
  1: fffff801`0bf9b1c0 @!"ECHO!EchoEvtDeviceAdd"
```

   Note
   You can use different syntax in conjunction with setting variables like <module>!<symbol>, <class>::<method>,'<file.cpp>:<line number>', or skip a number of times <condition> <#>. For more information, see Conditional breakpoints in WinDbg and other Windows debuggers.

6. List the current breakpoints to confirm that the breakpoint was set by typing the **bl** command.

```
0: kd> bl
1 e fffff801`0bf9b1c0     0001 (0001) ECHO!EchoEvtDeviceAdd
```

   The "e" in the output shown above indicates that the breakpoint number 1 is enabled to fire.

7. Restart code execution on the target system by typing the **go** command **g**.
8. **-> On the target system**

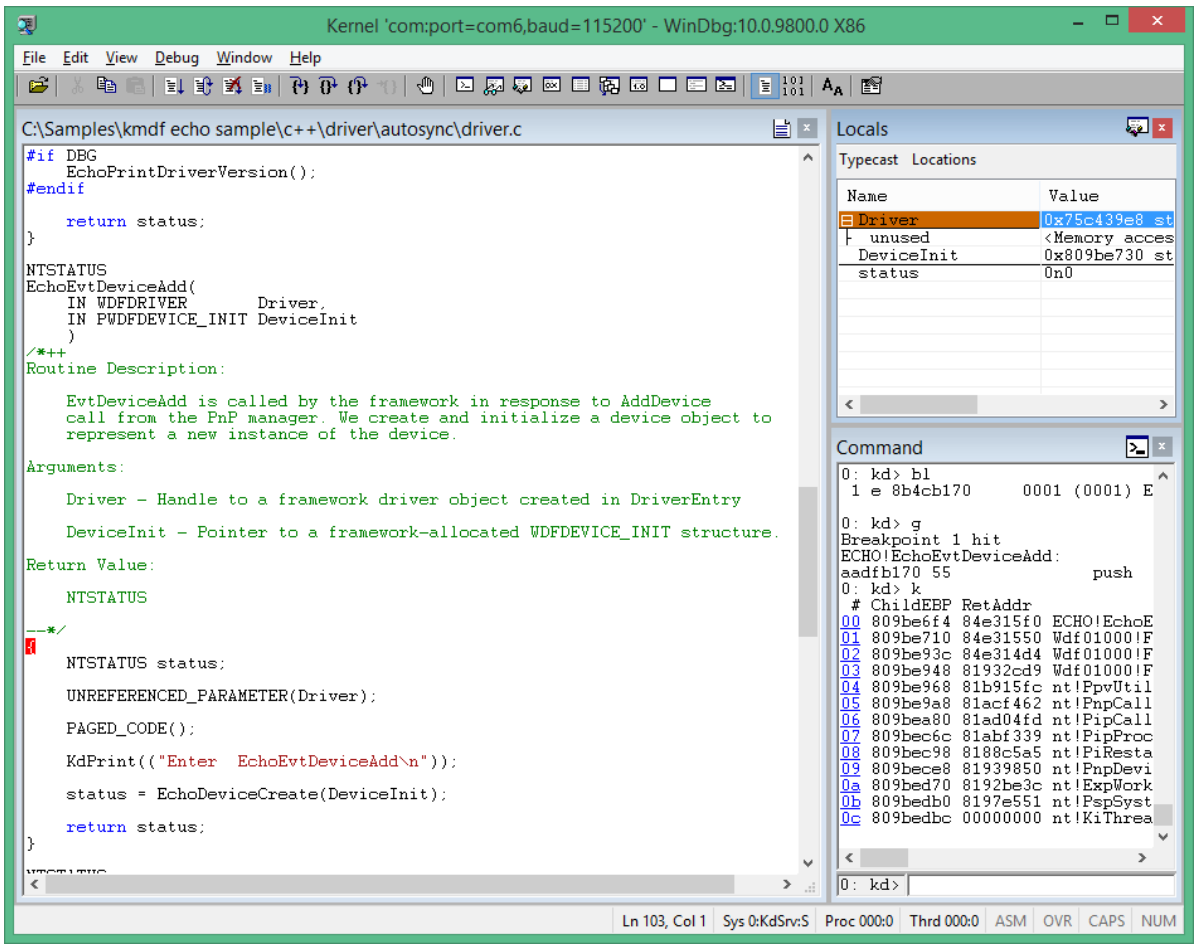
   In Windows, open **Device Manager** using the icon or by entering **mmc devmgmt.msc**. In Device Manager, expand the **Samples** node.

9. Right-click the KMDF Echo driver entry and select **Disable** from the menu.

10. Right-click the KMDF Echo driver entry again and select **Enable** from the menu.

11. **<- On the host system**

    When the driver is enabled, the *AddDevice* debug breakpoint should fire, and the execution of the driver code on the target system should halt. When the breakpoint is hit, the execution should be stopped at the start of the *AddDevice* routine. The debug command output will display "Breakpoint 1 hit".



12. Step through the code line-by-line by typing the **p** command or pressing F10 until you reach the following end of the *AddDevice* routine. The Brace character "}" will be highlighted as shown.

```
--*/
{
    NTSTATUS status;

    UNREFERENCED_PARAMETER(Driver);

    PAGED_CODE();

    KdPrint(("Enter  EchoEvtDeviceAdd\n"));

    status = EchoDeviceCreate(DeviceInit);

    return status;
}
```

13. In the next section, we will examine the state of the variables after the DeviceAdd code has executed.

Note
Modifying breakpoint state

You can modify existing breakpoints by using the following commands:

bl	Lists breakpoints.

bc	Clears a breakpoint from the list. Use bc * to clear all breakpoints.
bd	Disables a breakpoint. Use bd * to disable all breakpoints.
be	Enables a breakpoint. Use be * to enable all breakpoints.

Alternatively, you can also modify breakpoints by clicking **edit** > **breakpoints** in WinDbg. Note that the breakpoint dialog box only works with existing breakpoints. New breakpoints must be set from the command line.

Note
Setting memory access breakpoints

You can also set breakpoints that fire when a memory location is accessed. Use the **ba** (break on access) command, with the following syntax.

```
ba <access> <size> <address> {options}
```

OPTION	DESCRIPTION
e	execute (when CPU fetches an instruction from the address)
r	read/write (when CPU reads or writes to the address)
w	write (when the CPU writes to the address)

Note that you can only set four data breakpoints at any given time and it is up to you to make sure that you are aligning your data correctly or you won't trigger the breakpoint (words must end in addresses divisible by 2, dwords must be divisible by 4, and quadwords by 0 or 8).

For example, to set a read/write breakpoint on a specific memory address, you could use a command like this.

```
ba r 4 0x0003f7bf0
```

Note
Stepping through code from the Debugger Command window

The following are the commands that you can use to step through your code (with the associated keyboard short cuts shown in parentheses).

- Break in (Ctrl+Break) - This command will interrupt a system as long as the system is running and is in communication with WinDbg (the sequence in the Kernel Debugger is Ctrl+C).
- Run to cursor (F7 or Ctrl+F10) – Place the cursor in a source or disassembly window where you want the execution to break, then press F7; code execution will run to that point. Note that if the flow of code execution does not reach the point indicated by the cursor (an IF statement isn't executed), WinDbg would not break, because the code execution did not reach the indicated point.
- Run (F5) – Run until a breakpoint is encountered or an event like a bug check occurs.

- Step over (F10) – This command causes code execution to proceed one statement or one instruction at a time. If a call is encountered, code execution passes over the call without entering the called routine. (If the programming language is C or C++ and WinDbg is in source mode, source mode can be turned on or off using **Debug**>**Source Mode**).
- Step in (F11) – This command is like step-over, except that the execution of a call does go into the called routine.
- Step out (Shift+F11) – This command causes execution to run to and exit from the current routine (current place in the call stack). This is useful if you've seen enough of the routine.

For more information, see Source Code Debugging in WinDbg in the debugging reference documentation.

Section 8: Viewing variables and call stacks

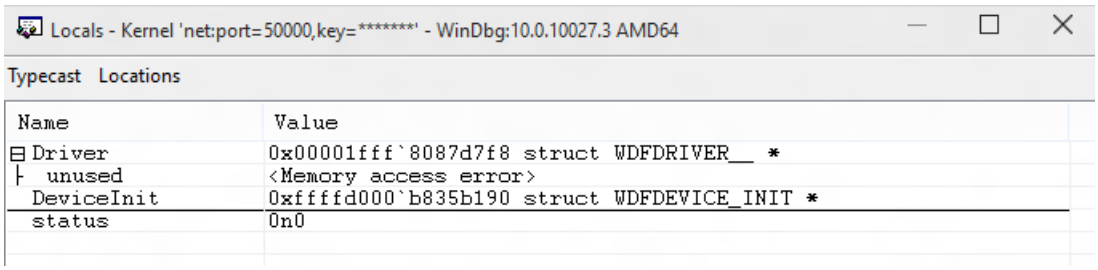
In Section 8, you will display information about variables and call stacks.

This lab assumes that you are stopped at the *AddDevice* routine using the process described earlier. To view the output show here, repeat the steps described previously, if necessary.

<- On the host system

Display variables

Use the **view**> **local** menu item to display local variables.



Global variables

You can find the location of a global variable address by typing *? <variable name>*.

Local variables

You can display the names and values of all local variables for a given frame by typing the **dv** command.

```
0: kd> dv
         Driver = 0x00001fff`7ff9c838
     DeviceInit = 0xffffd001`51978190
         status = 0n0
```

Callstacks

Note
The call stack is the chain of function calls that have led to the current location of the program counter. The top function on the call stack is the current function, and the next function is the function that called the current function, and so on.

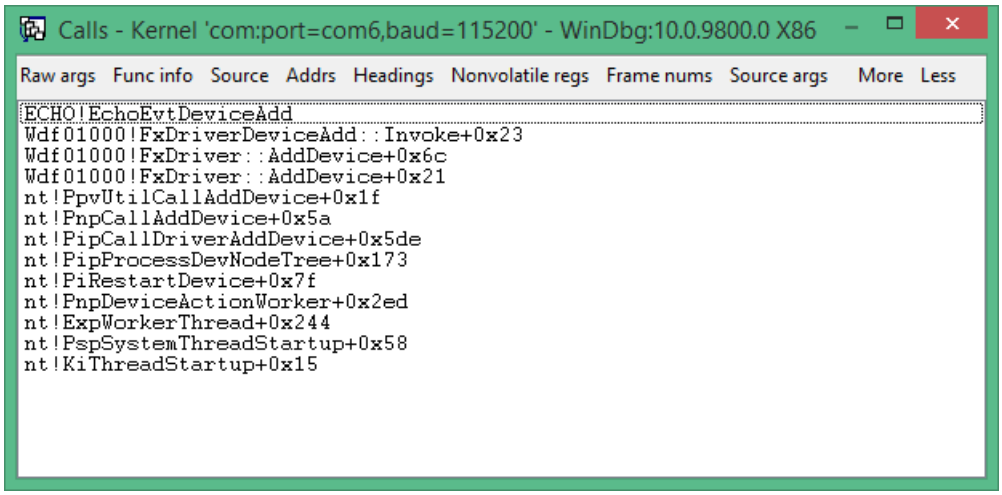
To display the call stack, use the k* commands.

kb	Displays the stack and first three parameters.

kp	Displays the stacks and the full list of parameters.
kn	Allows you to see the stack with the frame information next to it.

<-On the host system

1. If you want to keep the call stack available, you can click **view** > **call stack** to view it. Click on the columns at the top of the window to toggle the display of additional information.



2. Use the **kn** command to show the call stack while debugging the sample adapter code in a break state.

```
3: kd> kn
# Child-SP          RetAddr           Call Site
00 ffffd001`51978110 fffff801`0942f55b ECHO!EchoEvtDeviceAdd+0x66 [c:\Samples\kmdf echo
sample\c++\driver\autosync\driver.c @ 138]
01 (Inline Function) --------`-------- Wdf01000!FxDriverDeviceAdd::Invoke+0x30
[d:\wbrtm\minkernel\wdf\framework\shared\inc\private\common\fxdrivercallbacks.hpp @ 61]
02 ffffd001`51978150 fffff801`eed8097d Wdf01000!FxDriver::AddDevice+0xab
[d:\wbrtm\minkernel\wdf\framework\shared\core\km\fxdriverkm.cpp @ 72]
03 ffffd001`51978570 fffff801`ef129423 nt!PpvUtilCallAddDevice+0x35
[d:\9142\minkernel\ntos\io\pnpmgr\verifier.c @ 104]
04 ffffd001`519785b0 fffff801`ef0c4112 nt!PnpCallAddDevice+0x63 [d:\9142\minkernel\ntos\io\pnpmgr\enum.c @
7397]
05 ffffd001`51978630 fffff801`ef0c344f nt!PipCallDriverAddDevice+0x6e2
[d:\9142\minkernel\ntos\io\pnpmgr\enum.c @ 3390]
...
```

The call stack shows that the kernel (nt) called into Plug and Play code (PnP), that called driver framework code (WDF) that subsequently called the echo driver **DeviceAdd** function.

Section 9: Displaying processes and threads

Processes

In Section 9, you will display information about the process and threads running in kernel mode.

Note

You can display or set process information by using the **!process** debugger extension. We will set a breakpoint to examine the process that are used when a sound is played.

1. **<- On the host system**

Type the **dv** command to examine the locale variables associated with the **EchoEvtIo** routine as shown.

```
0: kd> dv ECHO!EchoEvtIo*
ECHO!EchoEvtIoQueueContextDestroy
ECHO!EchoEvtIoWrite
ECHO!EchoEvtIoRead
```

2. Clear the previous breakpoints using **bc ***.

```
0: kd> bc *
```

3. c. Set a symbol breakpoint on the **EchoEvtIo** routines using the following command.

```
0: kd> bm ECHO!EchoEvtIo*
  2: aade5490          @!"ECHO!EchoEvtIoQueueContextDestroy"
  3: aade55d0          @!"ECHO!EchoEvtIoWrite"
  4: aade54c0          @!"ECHO!EchoEvtIoRead"
```

4. List the breakpoints to confirm that the breakpoint is set properly.

```
0: kd> bl
1 e aabf0490 [c:\Samples\kmdf echo sample\c++\driver\autosync\queue.c @ 197]    0001 (0001)
ECHO!EchoEvtIoQueueContextDestroy
...
```

5. Type **g** to restart code execution.

```
0: kd> g
```

6. **-> On the target system**

   Run the EchoApp.exe driver test program on the target system.

7. **<- On the host system**

   When the test app runs, the I/O routine in the driver will be called. This will cause the breakpoint to fire, and execution of the driver code on the target system will halt.

```
Breakpoint 2 hit
ECHO!EchoEvtIoWrite:
fffff801`0bf95810 4c89442418      mov     qword ptr [rsp+18h],r8
```

8. Use the **!process** command to display the current process that is involved in running echoapp.exe.

```
0: kd> !process
PROCESS ffffe0007e6a7780
    SessionId: 1  Cid: 03c4    Peb: 7ff7cfec4000  ParentCid: 0f34
    DirBase: 1efd1b000  ObjectTable: ffffc001d77978c0  HandleCount:  34.
    Image: echoapp.exe
    VadRoot ffffe000802c79f0 Vads 30 Clone 0 Private 135. Modified 5. Locked 0.
    DeviceMap ffffc001d83c6e80
    Token                             ffffc001cf270050
    ElapsedTime                       00:00:00.052
    UserTime                          00:00:00.000
    KernelTime                        00:00:00.000
    QuotaPoolUsage[PagedPool]         33824
    QuotaPoolUsage[NonPagedPool]      4464
    Working Set Sizes (now,min,max)  (682, 50, 345) (2728KB, 200KB, 1380KB)
    PeakWorkingSetSize                652
    VirtualSize                       16 Mb
    PeakVirtualSize                   16 Mb
    PageFaultCount                    688
    MemoryPriority                    BACKGROUND
    BasePriority                      8
    CommitCharge                      138

        THREAD ffffe00080e32080  Cid 03c4.0ec0  Teb: 00007ff7cfece000 Win32Thread: 0000000000000000
RUNNING on processor 1
```

The output shows that the process is associated with the echoapp.exe which was running when our breakpoint on the driver write event was hit. For more information, see **!process**.

9. Use the **!process 0 0** to display summary information for all processes. In the output, use CTRL+F to locate the same process address for the process associated with the echoapp.exe image. In the example shown below, the process address is ffffe0007e6a7780.

```
...

PROCESS ffffe0007e6a7780
    SessionId: 1  Cid: 0f68    Peb: 7ff7cfe7a000  ParentCid: 0f34
    DirBase: 1f7fb9000  ObjectTable: ffffc001cec82780  HandleCount:  34.
    Image: echoapp.exe

...
```

10. Record the process ID associated with echoapp.exe to use later in this lab. You can also use CTRL+C, to copy the address to the copy buffer for later use.

    _____(echoapp.exe process address)

11. Enter **g** as required into the debugger to run the code forward until the echoapp.exe finishes running. It will hit the breakpoint in the read and write event a number of times. When echoapp.exe finishes, break in to the debugger, by pressing CTRL+ScrLk (Ctrl+Break).

12. Use the **!process** command to confirm that you are now running a different process. In the output shown below, the process with the Image value of *System* is different from the *Echo* Image value.

```
1: kd> !process
PROCESS ffffe0007b65d900
    SessionId: none  Cid: 0004    Peb: 00000000  ParentCid: 0000
    DirBase: 001ab000  ObjectTable: ffffc001c9a03000  HandleCount: 786.
    Image: System
    VadRoot ffffe0007ce45930 Vads 14 Clone 0 Private 22. Modified 131605. Locked 64.
    DeviceMap ffffc001c9a0c220
    Token                             ffffc001c9a05530
    ElapsedTime                       21:31:02.516
...
```

The output above shows that a system process ffffe0007b65d900 was running, when we stopped the OS.

13. Now, use the **!process** command to try to look at the process ID that had been associated with echoapp.exe that you recorded earlier. Provide your echoapp.exe process address that you recorded earlier, instead of the example process address shown below.

```
0: kd> !process ffffe0007e6a7780
TYPE mismatch for process object at 82a9acc0
```

The process object is no longer available, as the echoapp.exe process is no longer running.

Threads

Note

The commands to view and set threads are very similar to those of processes. Use the **!thread** command to view threads. Use **.thread** to set the current threads.

1. **<- On the host system**

   Enter **g** into the debugger to restart code execution on the target system.

2. **-> On the target system**

   Run the EchoApp.exe driver test program on the target system.

3. **<- On the host system**

   The breakpoint will be hit and code execution will halt.

```
Breakpoint 4 hit
ECHO!EchoEvtIoRead:
aade54c0 55              push    ebp
```

4. To view the threads that are running, type **!thread**. Information similar to the following should be displayed:

```
0: kd>  !thread
THREAD ffffe000809a0880  Cid 0b28.1158  Teb: 00007ff7d00dd000 Win32Thread: 0000000000000000 RUNNING on
processor 0
IRP List:
    ffffe0007bc5be10: (0006,01f0) Flags: 00060a30  Mdl: 00000000
Not impersonating
DeviceMap                 ffffc001d83c6e80
Owning Process            ffffe0008096c900       Image:         echoapp.exe
...
```

Note the image name of *echoapp.exe*, indicating that we are looking at the thread associated with the test app.

5. d. Use the **!process** command to determine if this is the only thread running in the process associated with echoapp.exe. Note that the thread number of the running thread in the process is the same thread running that the !thread command displayed.

```
0: kd> !process
PROCESS ffffe0008096c900
    SessionId: 1  Cid: 0b28    Peb: 7ff7d00df000  ParentCid: 0f34
    DirBase: 1fb746000  ObjectTable: ffffc001db6b52c0  HandleCount:  34.
    Image: echoapp.exe
    VadRoot ffffe000800cf920 Vads 30 Clone 0 Private 135. Modified 8. Locked 0.
    DeviceMap ffffc001d83c6e80
    Token                             ffffc001cf5dc050
    ElapsedTime                       00:00:00.048
    UserTime                          00:00:00.000
    KernelTime                        00:00:00.000
    QuotaPoolUsage[PagedPool]         33824
    QuotaPoolUsage[NonPagedPool]      4464
    Working Set Sizes (now,min,max)  (681, 50, 345) (2724KB, 200KB, 1380KB)
    PeakWorkingSetSize                651
    VirtualSize                       16 Mb
    PeakVirtualSize                   16 Mb
    PageFaultCount                    686
    MemoryPriority                    BACKGROUND
    BasePriority                      8
    CommitCharge                      138

        THREAD ffffe000809a0880  Cid 0b28.1158  Teb: 00007ff7d00dd000 Win32Thread: 0000000000000000
RUNNING on processor 0
```

6. Use the **!process 0 0 command** to locate the process address of two related processes and record those process address here.

   Cmd.exe: _____

   EchoApp.exe: _____

```
0: kd> !process 0 0

…

PROCESS ffffe0007bbde900
    SessionId: 1  Cid: 0f34    Peb: 7ff72dfa7000  ParentCid: 0c64
    DirBase: 19c5fa000  ObjectTable: ffffc001d8c2f300  HandleCount:  31.
    Image: cmd.exe
…
PROCESS ffffe0008096c900
    SessionId: 1  Cid: 0b28    Peb: 7ff7d00df000  ParentCid: 0f34
    DirBase: 1fb746000  ObjectTable: ffffc001db6b52c0  HandleCount:  34.
    Image: echoapp.exe
…
```

   Note You can alternatively use **!process 0 17** to display detailed information about every process. The output from this command can be lengthy. The output can be searched using Ctrl+F.

7. Use the **!process** command to list process information for both processes running your PC. Provide the process address from your **!process 0 0** output, not the address shown below.

   This example output is for the cmd.exe process ID that was recorded earlier. Note that the image name for this process ID is cmd.exe.

```
0: kd>  !process ffffe0007bbde900
PROCESS ffffe0007bbde900
    SessionId: 1  Cid: 0f34    Peb: 7ff72dfa7000  ParentCid: 0c64
    DirBase: 19c5fa000  ObjectTable: ffffc001d8c2f300  HandleCount:  31.
    Image: cmd.exe
    VadRoot ffffe0007bb8e7b0 Vads 25 Clone 0 Private 117. Modified 20. Locked 0.
    DeviceMap ffffc001d83c6e80
    Token                             ffffc001d8c48050
    ElapsedTime                       21:33:05.840
    UserTime                          00:00:00.000
    KernelTime                        00:00:00.000
    QuotaPoolUsage[PagedPool]         24656
    QuotaPoolUsage[NonPagedPool]      3184
    Working Set Sizes (now,min,max)  (261, 50, 345) (1044KB, 200KB, 1380KB)
    PeakWorkingSetSize                616
    VirtualSize                       2097164 Mb
    PeakVirtualSize                   2097165 Mb
    PageFaultCount                    823
    MemoryPriority                    FOREGROUND
    BasePriority                      8
    CommitCharge                      381

        THREAD ffffe0007cf34880  Cid 0f34.0f1c  Teb: 00007ff72dfae000 Win32Thread: 0000000000000000
WAIT: (UserRequest) UserMode Non-Alertable
            ffffe0008096c900  ProcessObject
        Not impersonating
...
```

This example output is for the echoapp.exe process ID that was recorded earlier.

```
0: kd>  !process ffffe0008096c900
PROCESS ffffe0008096c900
    SessionId: 1  Cid: 0b28    Peb: 7ff7d00df000  ParentCid: 0f34
    DirBase: 1fb746000  ObjectTable: ffffc001db6b52c0  HandleCount:  34.
    Image: echoapp.exe
    VadRoot ffffe000800cf920 Vads 30 Clone 0 Private 135. Modified 8. Locked 0.
    DeviceMap ffffc001d83c6e80
    Token                             ffffc001cf5dc050
    ElapsedTime                       00:00:00.048
    UserTime                          00:00:00.000
    KernelTime                        00:00:00.000
    QuotaPoolUsage[PagedPool]         33824
    QuotaPoolUsage[NonPagedPool]      4464
    Working Set Sizes (now,min,max)  (681, 50, 345) (2724KB, 200KB, 1380KB)
    PeakWorkingSetSize                651
    VirtualSize                       16 Mb
    PeakVirtualSize                   16 Mb
    PageFaultCount                    686
    MemoryPriority                    BACKGROUND
    BasePriority                      8
    CommitCharge                      138

        THREAD ffffe000809a0880  Cid 0b28.1158  Teb: 00007ff7d00dd000 Win32Thread: 0000000000000000
RUNNING on processor 0
        IRP List:
            ffffe0007bc5be10: (0006,01f0) Flags: 00060a30  Mdl: 00000000
        Not impersonating
...
```

8. Record the first thread address associated with the two processes here.

   Cmd.exe: _____

   EchoApp.exe: _____

9. Use the **!Thread** command to display information about the current thread.

```
0: kd>  !Thread
THREAD ffffe000809a0880  Cid 0b28.1158  Teb: 00007ff7d00dd000 Win32Thread: 0000000000000000 RUNNING on
processor 0
IRP List:
    ffffe0007bc5be10: (0006,01f0) Flags: 00060a30  Mdl: 00000000
Not impersonating
DeviceMap                 ffffc001d83c6e80
Owning Process            ffffe0008096c900       Image:         echoapp.exe
Attached Process          N/A            Image:         N/A
...
```

As expected, the current thread is the thread associated with echoapp.exe and it is in a running state.

10. Use the **!Thread** command to display information about the thread associated with cmd.exe process. Provide the thread address you recorded earlier.

```
0: kd> !Thread ffffe0007cf34880
THREAD ffffe0007cf34880  Cid 0f34.0f1c  Teb: 00007ff72dfae000 Win32Thread: 0000000000000000 WAIT:
(UserRequest) UserMode Non-Alertable
    ffffe0008096c900  ProcessObject
Not impersonating
DeviceMap                 ffffc001d83c6e80
Owning Process            ffffe0007bbde900       Image:         cmd.exe
Attached Process          N/A            Image:         N/A
Wait Start TickCount      4134621        Ticks: 0
Context Switch Count      4056           IdealProcessor: 0
UserTime                  00:00:00.000
KernelTime                00:00:01.421
Win32 Start Address 0x00007ff72e9d6e20
Stack Init ffffd0015551dc90 Current ffffd0015551d760
Base ffffd0015551e000 Limit ffffd00155518000 Call 0
Priority 14 BasePriority 8 UnusualBoost 3 ForegroundBoost 2 IoPriority 2 PagePriority 5
Child-SP          RetAddr           : Args to Child
: Call Site
ffffd001`5551d7a0 fffff801`eed184fe : fffff801`eef81180 ffffe000`7cf34880 00000000`ffffffe
00000000`ffffffe : nt!KiSwapContext+0x76 [d:\9142\minkernel\ntos\ke\amd64\ctxswap.asm @ 109]
ffffd001`5551d8e0 fffff801`eed17f79 : ffff03a5`ca56a3c8 000000de`b6a6e990 000000de`b6a6e990
00007ff7`d00df000 : nt!KiSwapThread+0x14e [d:\9142\minkernel\ntos\ke\thredsup.c @ 6347]
ffffd001`5551d980 fffff801`eecea340 : ffffd001`5551da18 00000000`00000000 00000000`00000000
00000000`00000388 : nt!KiCommitThreadWait+0x129 [d:\9142\minkernel\ntos\ke\waitsup.c @ 619]
...
```

This thread is associated with cmd.exe and is in a wait state.

11. Provide the thread address of the waiting CMD.exe thread to change the context to that waiting thread.

```
0: kd> .Thread ffffe0007cf34880
Implicit thread is now ffffe000`7cf34880
```

12. Use the **k** command to view the call stack associated with the waiting thread.

```
0: kd> k
  *** Stack trace for last set context - .thread/.cxr resets it
# Child-SP          RetAddr           Call Site
00 ffffd001`5551d7a0 fffff801`eed184fe nt!KiSwapContext+0x76
[d:\9142\minkernel\ntos\ke\amd64\ctxswap.asm @ 109]
01 ffffd001`5551d8e0 fffff801`eed17f79 nt!KiSwapThread+0x14e [d:\9142\minkernel\ntos\ke\thredsup.c @
6347]
02 ffffd001`5551d980 fffff801`eecea340 nt!KiCommitThreadWait+0x129 [d:\9142\minkernel\ntos\ke\waitsup.c
@ 619]
03 ffffd001`5551da00 fffff801`ef02e642 nt!KeWaitForSingleObject+0x2c0 [d:\9142\minkernel\ntos\ke\wait.c
@ 683]
...
```

Call stack elements such as **KiCommitThreadWait** indicate that this thread is not running as is expected.

Note
For more information about threads and processes, see the following references:

Threads and Processes

Changing Contexts

Section 10: IRQL, Registers and Ending the WinDbg session

Viewing the saved IRQL

In Section 10, you will display the IRQL, and the contents of the regsisters.

<- On the host system

The interrupt request level (IRQL) is used to manage the priority of interrupt servicing. Each processor has an IRQL setting that threads can raise or lower. Interrupts that occur at or below the processor's IRQL setting are masked and will not interfere with the current operation. Interrupts that occur above the processor's IRQL setting take precedence over the current operation. The **!irql** extension displays the interrupt request level (IRQL) on the current processor of the target computer before the debugger break occurred. When the target computer breaks into the debugger, the IRQL changes, but the IRQL that was effective just before the debugger break is saved and is displayed by **!irql**.

```
0: kd> !irql
Debugger saved IRQL for processor 0x0 -- 2 (DISPATCH_LEVEL)
```

Viewing the registers

<-On the host system

Display the contents of the registers for the current thread on the current processor by using the **r (Registers)** command.

```
0: kd> r
rax=000000000000c301 rbx=ffffe00173eed880 rcx=0000000000000001
rdx=000000d800000000 rsi=ffffe00173eed8e0 rdi=ffffe00173eed8f0
rip=fffff803bb757020 rsp=ffffd001f01f8988 rbp=ffffe00173f0b620
 r8=000000000000003e  r9=ffffe00167a4a000 r10=000000000000001e
r11=ffffd001f01f88f8 r12=0000000000000000 r13=ffffd001f01efdc0
r14=0000000000000001 r15=0000000000000000
iopl=0         nv up ei pl nz na pe nc
cs=0010  ss=0018  ds=002b  es=002b  fs=0053  gs=002b             efl=00000202
nt!DbgBreakPointWithStatus:
fffff803`bb757020 cc              int     3
```

Alternatively, you can display the contents of the registers by clicking **view** > **registers**. For more information see **r (Registers)**.

Viewing the contents of the registers can be helpful when stepping through assembly language code execution and in other scenarios. For more information about assembly language disassembly, see Annotated x86 Disassembly and Annotated x64 Disassembly.

For information about contents of the register, see x86 Architecture and x64 Architecture.

Ending the WinDbg session

<-On the host system

To end a user-mode debugging session, return the debugger to dormant mode, and set the target application to run again, enter the **qd** (Quit and Detach) command.

Be sure and use the **g** command to let the target computer run code, so that it can be used. It also a good idea to clear any break points using **bc ***, so that the target computer won't break and try to connect to the host computer debugger.

```
0: kd> qd
```

For more information, see Ending a Debugging Session in WinDbg in the debugging reference documentation.

Section 11: Windows debugging resources

Additional information is available on Windows debugging. Note that some of these books will use older versions of Windows such as Windows Vista in their examples, but the concepts discussed are applicable to most versions of Windows.

Books

- Advanced Windows Debugging by Mario Hewardt and Daniel Pravat
- Inside Windows Debugging: A Practical Guide to Debugging and Tracing Strategies in Windows® by Tarik Soulami
- Windows Internals by Mark E. Russinovich, David A. Solomon and Alex Ionescu

Video

The Defrag Tools Show WinDbg Episodes 13-29 https://channel9.msdn.com/Shows/Defrag-Tools

Training Vendors:

OSR https://www.osr.com/

Related topics

Standard Debugging Techniques

Specialized Debugging Techniques

Getting Started with Windows Debugging

Debug Drivers - Step by Step Lab (Sysvad Kernel Mode)

6/25/2019 • 55 minutes to read • Edit Online

This lab provides hands-on exercises that demonstrate how to debug the Sysvad audio kernel-mode device driver.

Microsoft Windows Debugger (WinDbg) is a powerful Windows-based debugging tool that you can use to perform user-mode and kernel-mode debugging. WinDbg provides source-level debugging for the Windows kernel, kernel-mode drivers, and system services, as well as user-mode applications and drivers.

WinDbg can step through source code, set breakpoints, view variables (including C++ objects), stack traces, and memory. Its Debugger Command window allows the user to issue a wide variety of commands.

Lab setup

You will need the following hardware to be able to complete the lab:

- A laptop or desktop computer (host) running Windows 10
- A laptop or desktop computer (target) running Windows 10
- A network hub/router and network cables to connect the two PCs
- Access to the internet to download symbol files

You will need the following software to be able to complete the lab.

- Microsoft Visual Studio 2017
- Windows Software Development Kit (SDK) for Windows 10
- Windows Driver Kit (WDK) for Windows 10
- The sample Sysvad audio driver for Windows 10

For information on downloading and installing the WDK, see Download the Windows Driver Kit (WDK).

Sysvad debugging walkthrough

This lab walk you through the process of debugging a kernel-mode driver. The exercises use the Syvad virtual audio driver sample. Because the Syvad audio driver doesn't interact with actual audio hardware, it can be used on most devices. The lab covers the following tasks:

- Section 1: Connect to a kernel-mode WinDbg session
- Section 2: kernel-mode debugging commands and techniques
- Section 3: Download and build the Sysvad audio driver
- Section 4: Install the Sysvad audio driver on the target system
- Section 5: Use WinDbg to display information about the driver
- Section 6: Display Plug and Play device tree information
- Section 7: Work with breakpoints and source code
- Section 8: Look at variables
- Section 9: View call stacks
- Section 10: Display processes and threads
- Section 11: IRQL, registers and disassembly
- Section 12: Work with memory

- Section 13: Ending the WinDbg session
- Section 14: Windows debugging resources

Echo driver lab

The Echo driver is a simpler driver then the Sysvad audio driver. If you are new to WinDbg, you may want to consider first completing the Debug Universal Drivers - Step-by-Step Lab (Echo kernel mode). This lab reuses the setup directions from that lab, so if you have completed that lab you can skip sections 1 and 2 here.

Section 1: Connect to a kernel-mode WinDbg session

In Section 1, you will configure network debugging on the host and target system.

The PCs in this lab need to be configured to use an Ethernet network connection for kernel debugging.

This lab uses two computers. WinDbg runs on the *host* system and the Sysvad driver runs on the *target* system.

Use a network hub/router and network cables to connect the two PCs.

To work with kernel-mode applications and use WinDbg, we recommend that you use the KDNET over Ethernet transport. For information about how to use the Ethernet transport protocol, see Getting Started with WinDbg (Kernel-Mode). For more information about setting up the target computer, see Preparing a Computer for Manual Driver Deployment and Setting Up KDNET Network Kernel Debugging Automatically.

Configure kernel–mode debugging using ethernet

To enable kernel-mode debugging on the target system, perform the following steps.

<- On the host system

1. Open a command prompt on the host system and type **ipconfig /all** to determine its IP address.

```
C:\>ipconfig /all
Windows IP Configuration

 Host Name . . . . . . . . . . . . : TARGETPC
...

Ethernet adapter Ethernet:
   Connection-specific DNS Suffix  . :
   Link-local IPv6 Address . . . . . : fe80::c8b6:db13:d1e8:b13b3
   Autoconfiguration IPv4 Address. . : 169.182.1.1
   Subnet Mask . . . . . . . . . . . : 255.255.0.0
   Default Gateway . . . . . . . . . :
```

2. Record the IP address of the host System: _____

3. Record the Host Name of the host System: _____

-> On the target system

4. Open a command prompt on the target system and use the **ping** command to confirm network connectivity between the two systems. Use the actual IP address of the host system you recorded instead of 169.182.1.1 that is shown in the sample output.

```
C:\> ping 169.182.1.1

Pinging 169.182.1.1 with 32 bytes of data:
Reply from 169.182.1.1: bytes=32 time=1ms TTL=255
Reply from 169.182.1.1: bytes=32 time<1ms TTL=255
Reply from 169.182.1.1: bytes=32 time<1ms TTL=255
Reply from 169.182.1.1: bytes=32 time<1ms TTL=255

Ping statistics for 169.182.1.1:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 0ms, Maximum = 1ms, Average = 0ms
```

To use the KDNET utility to enable kernel-mode debugging on the target system, peform the following steps.

1. On the host system, locate the WDK KDNET directory. By default it is located here.

   C:\Program Files (x86)\Windows Kits\10\Debuggers\x64

> **NOTE**
>
> This labs assumes that both PCs are running a 64 bit version of Windowson both the target and host. If that is not the case, the best approach is to run the same "bitness" of tools on the host that the target is running. For example if the target is running 32 bit Windows, run a 32 version of the debugger on the host. For more information, see Choosing the 32-Bit or 64-Bit Debugging Tools.

2. Locate these two files and copy them to a network share or thumb drive, so that they will be available on the target computer.

   kdnet.exe

   VerifiedNICList.xml

3. On the target computer, open a Command Prompt window as Administrator. Enter this command to validate that the NIC on the target PC is suported.

```
C:\KDNET>kdnet

Network debugging is supported on the following NICs:
busparams=0.25.0, Intel(R) 82579LM Gigabit Network Connection, KDNET is running on this NIC.kdnet.exe
```

4. Type this command to set the IP address of the host system. Use the actual IP address of the host system you recorded instead of 169.182.1.1 that is shown in the sample output. Pick a unique port address for each target/host pair that you work with, such as 50010.

```
C:\>kdnet 169.182.1.1 50010

Enabling network debugging on Intel(R) 82577LM Gigabit Network Connection.
Key=2steg4fzbj2sz.23418vzkd4ko3.1g34ou07z4pev.1sp3yo9yz874p
```

> **IMPORTANT**
>
> Before using BCDEdit to change boot information you may need to temporarily suspend Windows security features such as BitLocker and Secure Boot on the test PC. Re-enable these security features when testing is complete and appropriately manage the test PC, when the security features are disabled.

5. Type this command to confirm that the dbgsettings are set properly.

```
C:\> bcdedit /dbgsettings
busparams               0.25.0
key                     2steg4fzbj2sz.23418vzkd4ko3.1g34ou07z4pev.1sp3yo9yz874p
debugtype               NET
hostip                  169.182.1.1
port                    50010
dhcp                    Yes
The operation completed successfully.
```

Copy the auto generated unique key into a text file, to avoid having to type it in on the host PC. Copy the text file with the key over to the host system.

Note Firewalls and debuggers

If you receive a pop-up message from the firewall, and you wish to use the debugger, check **all three** of the boxes.

<- On the host system

1. On the host computer, open a Command Prompt window as Administrator. Change to the WinDbg.exe directory. We will use the x64version of WinDbg.exe from the Windows Driver Kit (WDK) that was installed as part of the Windows kit installation.

```
C:\> Cd C:\Program Files (x86)\Windows Kits\10\Debuggers\x64
```

2. Launch WinDbg with remote user debug using the following command. The value for the key and port match what you set earlier using BCDEdit on the target.

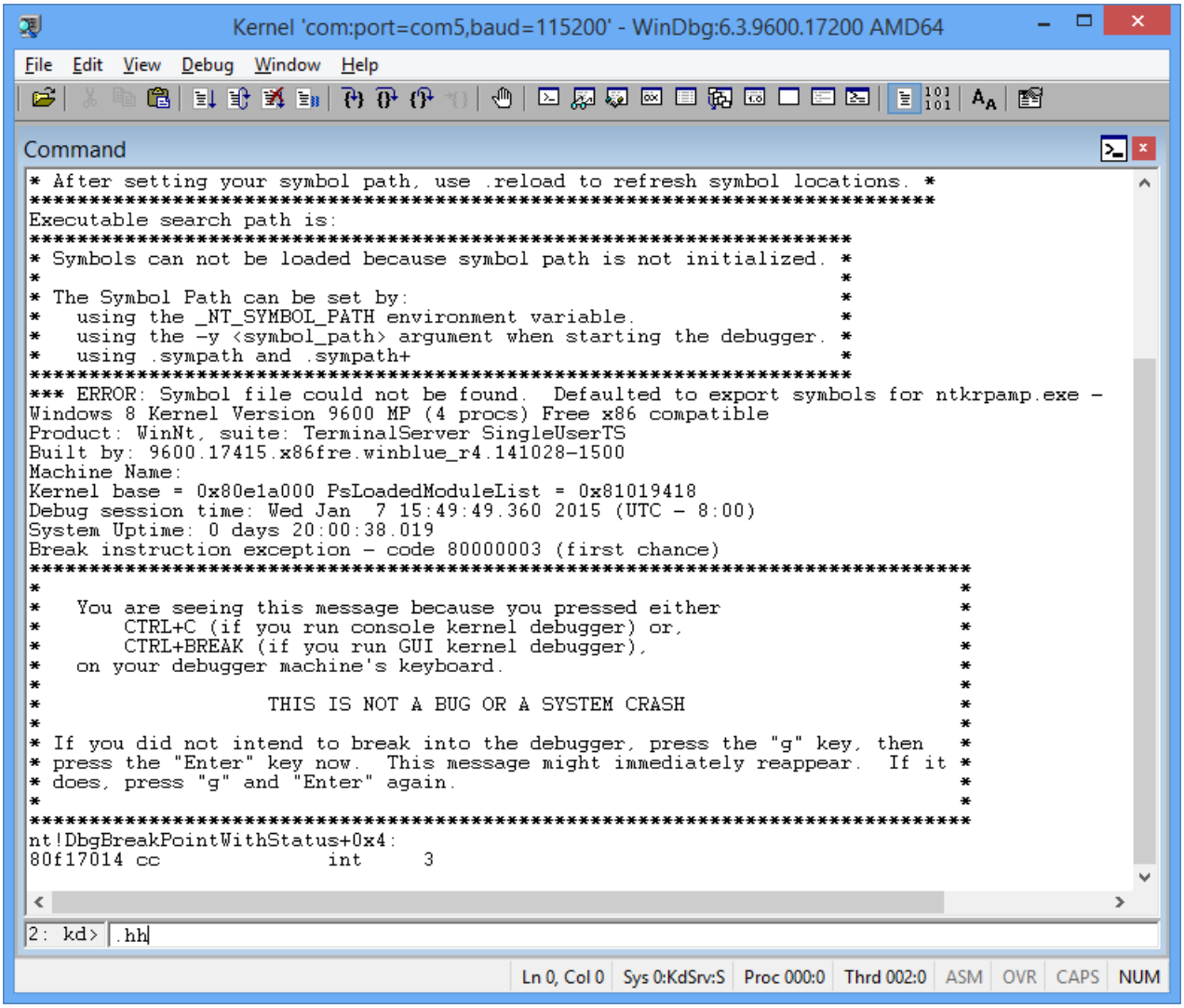
```
C:\> WinDbg -k net:port=50010,key=2steg4fzbj2sz.23418vzkd4ko3.1g34ou07z4pev.1sp3yo9yz874p
```

->On the target system

Reboot the target system.

<-On the host system

In a minute or two, debug output should be displayed on the host system.



The Debugger Command window is the primary debugging information window in WinDbg. You can enter debugger commands and view the command output in this window.

The Debugger Command window is split into two panes. You type commands in the smaller pane (the command entry pane) at the bottom of the window and view the command output in the larger pane at the top of the window.

In the command entry pane, use the up arrow and down arrow keys to scroll through the command history. When a command appears, you can edit it or press **ENTER** to run the command.

Section 2: kernel-mode debugging commands and techniques

In Section 2, you will use debug commands to display information about the target system.

<- On the host system

Enable Debugger Markup Language (DML) with .prefer_dml

Some debug commands display text using Debugger Markup Language that you can click on to quickly gather more information.

1. Use Ctrl+Break (Scroll Lock) in WinDBg to break into the code running on the target system. It may take a bit of time for the target system to respond.
2. Type the following command to enable DML in the Debugger Command window.

```
0: kd> .prefer_dml 1
DML versions of commands on by default
```

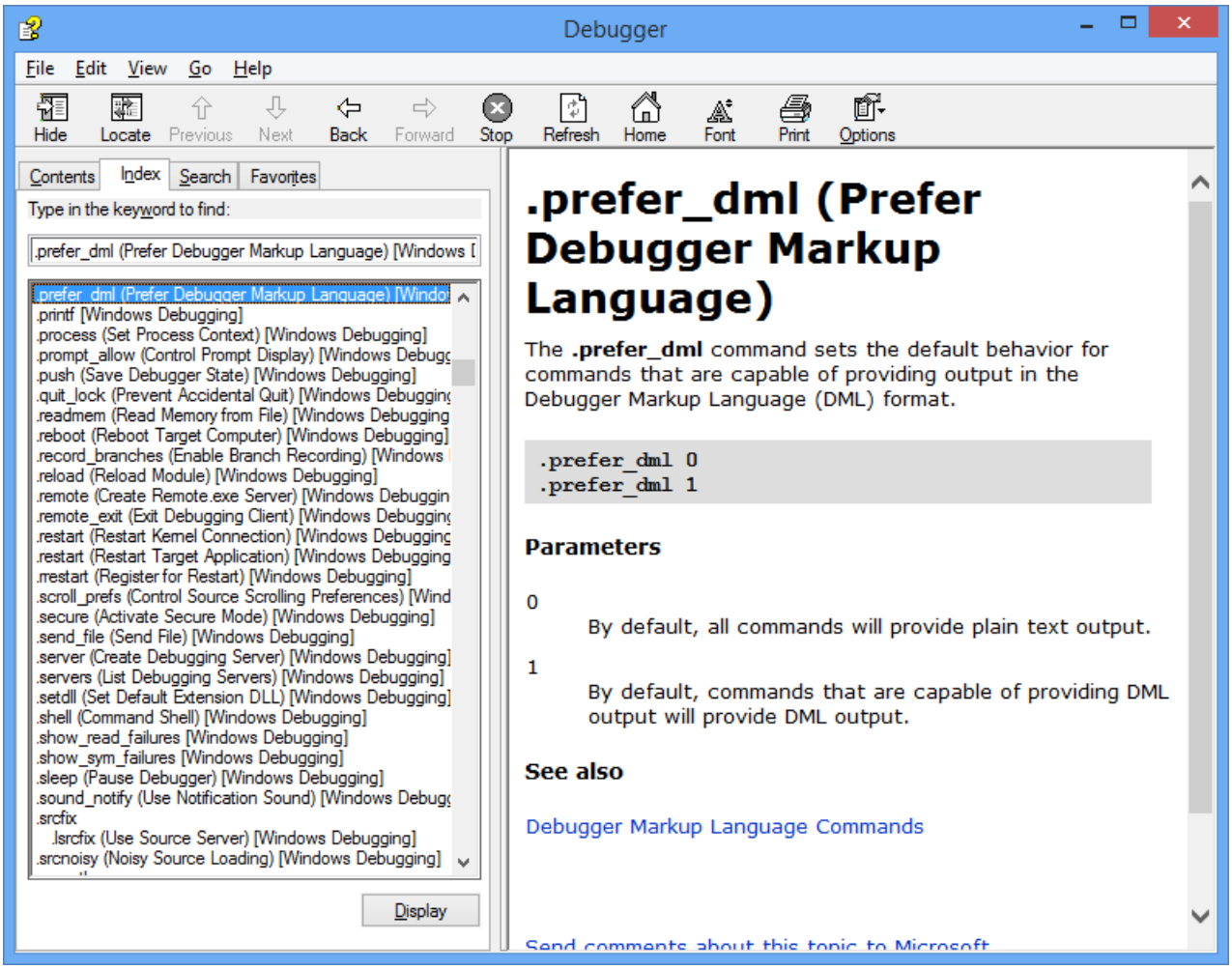
Use .hh to get help

You can access reference command help using the **.hh** command.

3. Type the following command to view the command reference help for **.prefer_dml**.

```
0: kd> .hh .prefer_dml
```

The Debugger help file will display help for the **.prefer_dml** command.



Display the version of Windows on the target system

5. Display detailed version information on the target system by typing the **vertarget (Show Target Computer Version)** command in the WinDbg window.

```
0: kd> vertarget
Windows 10 Kernel Version 9926 MP (4 procs) Free x64
Product: WinNt, suite: TerminalServer SingleUserTS
Built by: 9926.0.amd64fre.fbl_awesome1501.150119-1648
Machine Name: ""
Kernel base = 0xfffff801`8d283000 PsLoadedModuleList = 0xfffff801`8d58aef0
Debug session time: Fri Feb 20 10:15:17.807 2015 (UTC - 8:00)
System Uptime: 0 days 01:31:58.931
```

List the loaded modules

6. You can verify that you are working with the right kernel-mode process by displaying the loaded modules by typing the **lm (List Loaded Modules)** command in the WinDbg window.

```
0: Kd> lm
start             end                 module name
fffff801`09200000 fffff801`0925f000   volmgrx    (no symbols)
fffff801`09261000 fffff801`092de000   mcupdate_GenuineIntel   (no symbols)
fffff801`092de000 fffff801`092ec000   werkernel   (export symbols)       werkernel.sys
fffff801`092ec000 fffff801`0934d000   CLFS       (export symbols)       CLFS.SYS
fffff801`0934d000 fffff801`0936f000   tm         (export symbols)       tm.sys
fffff801`0936f000 fffff801`09384000   PSHED      (export symbols)       PSHED.dll
fffff801`09384000 fffff801`0938e000   BOOTVID    (export symbols)       BOOTVID.dll
fffff801`0938e000 fffff801`093f7000   spaceport   (no symbols)
fffff801`09400000 fffff801`094cf000   Wdf01000   (no symbols)
fffff801`094d9000 fffff801`09561000   CI         (export symbols)       CI.dll
...
```

Note Output that has been omitted is indicated with "... " in this lab.

Because we have yet to set the symbol path and loaded symbols, limited information is available in the debugger.

Section 3: Download and build the Sysvad audio driver

In Section 3, you will download and build the Sysvad audio driver.

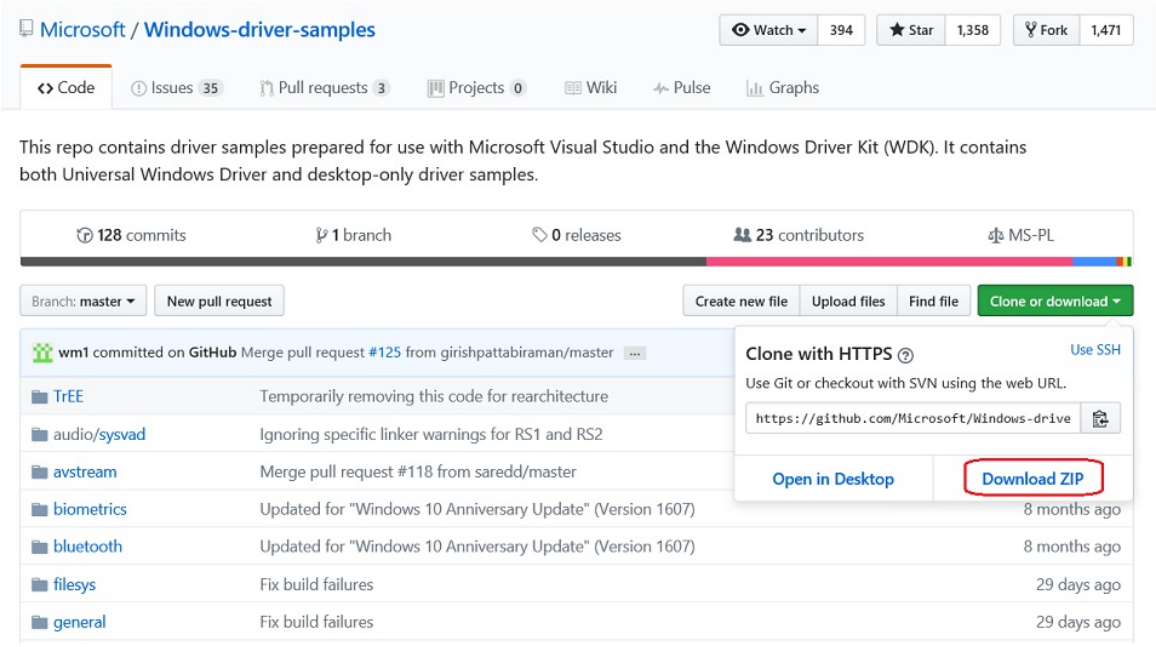
Typically, you would be working with your own driver code when you use WinDbg. To become familiar with debugging audio drivers, the Sysvad virtual audio sample driver is used. This sample is used to illustrate how you can single step through native kernel-mode code. This technique can be very valuable for debugging complex kernel-mode code issues.

To download and build the Sysvad sample audio driver, perform the following steps.

1. **Download and extract the Sysvad audio sample from GitHub**

   You can use a browser to view the Sysvad sample and Readme.md file here:

   https://github.com/Microsoft/Windows-driver-samples/tree/master/audio/sysvad



   This lab, shows how to download the universal driver samples in one zip file.

a. Download the master.zip file to your local hard drive.

https://github.com/Microsoft/Windows-driver-samples/archive/master.zip

b. Right-click *Windows-driver-samples-master.zip*, and choose **Extract All**. Specify a new folder, or browse to an existing one that will store the extracted files. For example, you could specify *C:\WDK_Samples* as the new folder into which the files are extracted.

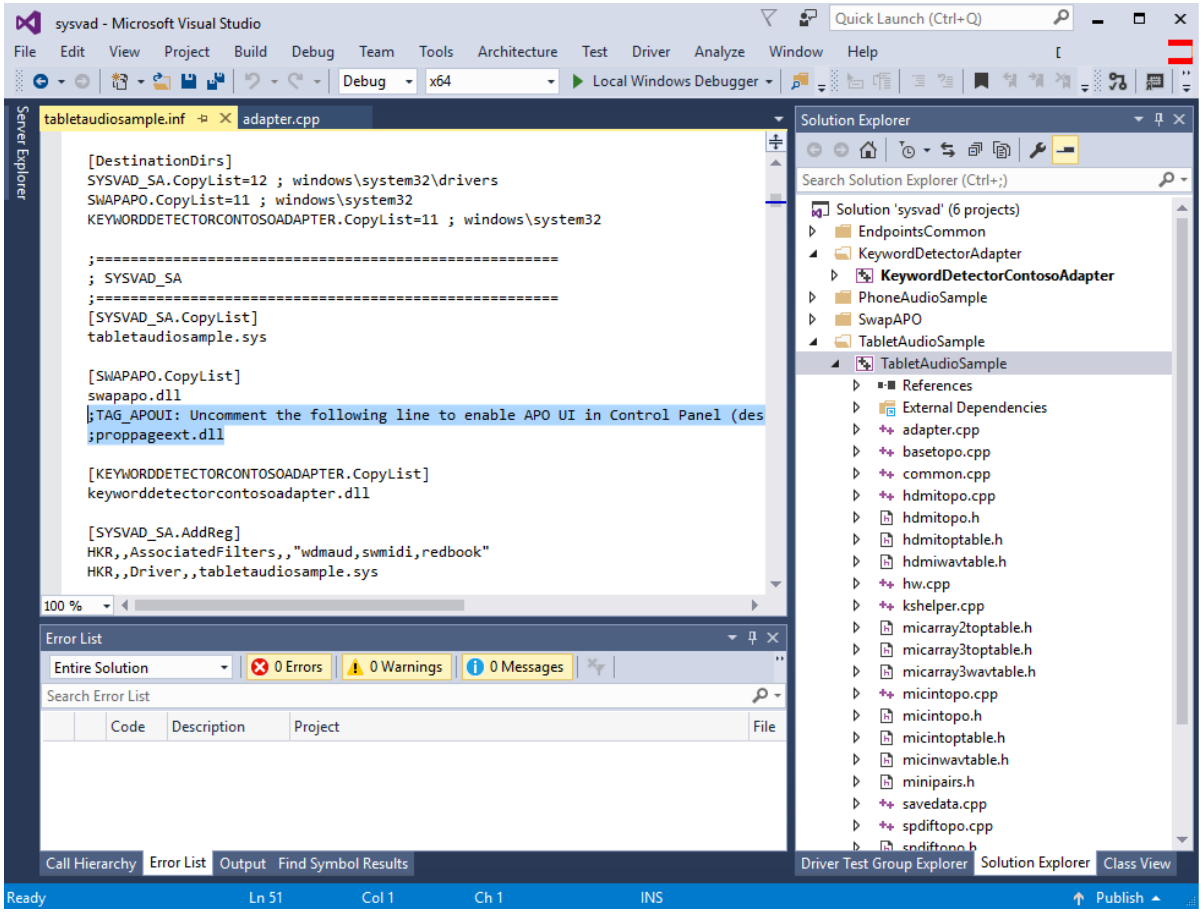
c. After the files are extracted, navigate to the following subfolder.

C:\WDK_Samples\Sysvad

2. **Open the driver solution in Visual Studio**

   In Visual Studio, click **File** > **Open** > **Project/Solution...** and navigate to the folder that contains the extracted files (for example, *C:\WDK_Samples\Sysvad*). Double-click the *Syvad* solution file.

   In Visual Studio locate the Solution Explorer. (If this is not already open, choose **Solution Explorer** from the **View** menu.) In Solution Explorer, you can see one solution that has a number of projects and what is included in the sample changes from time to time.



3. **Set the sample's configuration and platform**

   In Solution Explorer, right-click **Solution 'sysvad' (7 projects)**, and choose **Configuration Manager**. Make sure that the configuration and platform settings are the same for the four projects. By default, the configuration is set to "Win10 Debug", and the platform is set to "Win64" for all the projects. If you make any configuration and/or platform changes for one project, you must make the same changes for the remaining three projects.

   Note This lab assumes that 64 bit Windows is being used. If you are using 32 bit Windows, build the driver for 32 bit.

4. **Check driver signing**

Locate the TabletAudioSample. Open the Sysvad driver's property page and make sure **Driver Signing** > **Sign Mode** is set to *Test Sign*.

5. **Build the sample using Visual Studio**

   In Visual Studio, click **Build** > **Build Solution**.

   The build windows should display a message indicating that the build for all six projects succeeded.

6. **Locate the built driver files**

   In File Explorer, navigate to the folder that contains the extracted files for the sample. For example, you would navigate to *C:\WDK_Samples\Sysvad*, if that's the folder you specified earlier. Within that folder, the location of the compiled driver files varies depending on the configuration and platform settings that you selected in the **Configuration Manager**. For example, if you left the default settings unchanged, then the compiled driver files will be saved to a folder named *\x64\Debug* for a 64-bit, debug build.

   Navigate to the folder that contains the built files for the TabletAudioSample driver:

   C:\WDK_Samples\Sysvad\TabletAudioSample\x64\Debug. The folder will contain the TabletAudioSample .SYS driver, symbol pdp file and the inf file. You will also need to locate the SwapAPO, and KeywordDetectorContosoAdapter dlls and symbol files.

   To install the driver, you will need the following files.

TabletAudioSample.sys	The driver file.
TabletAudioSample.pdb	The driver symbol file.
tabletaudiosample.inf	An information (INF) file that contains information needed to install the driver.
KeywordDetectorContosoAdapter.dl	A sample keyword detector.
KeywordDetectorContosoAdapter.pdb	The sample keyword detector symbol file.
lSwapAPO.dll	A sample driver extension for a UI to manage APOs.
lSwapAPO.pdb	The APO UI symbol file.
TabletAudioSample.cer	The TabletAudioSample certificate file.

7. Locate a USB thumb drive or set up a network share to copy the built driver files from the host to the target system.

In the next section, you will copy the code to the target system, and install and test the driver.

Section 4: Install the Sysvad audio driver sample on the target system

In Section 4, you will use devcon to install the Sysvad audio driver.

-> On the target system

The computer where you install the driver is called the *target computer* or the *test computer*. Typically, this is a separate computer from the computer on which you develop and build the driver package. The computer where you develop and build the driver is called the *host computer*.

The process of moving the driver package to the target computer and installing the driver is called *deploying* the driver. You can deploy the sample Sysvad driver, automatically or manually.

Before you manually deploy a driver, you must prepare the target computer by turning on test signing. You also need to locate the DevCon tool in your WDK installation. After that you're ready to run the built driver sample on the target system.

To install the driver on the target system, perform the following steps.

1. **Enable test signed drivers**

   To enable the ability to run test signed drivers:

   a. Open Windows Settings.

   b. In **Update and Security**, select **Recovery**.

   c. Under **Advanced startup**, click **Restart Now**.

   d. When the PC restarts, select **Troubleshoot**.

   e. Then select **Advanced options**, **Startup Settings** and then click **Restart**.

   f. Select Disable driver signature enforcement by pressing the **F7** key.

   g. The PC will start with the new values in place.

2. **<- On the host system**

   Navigate to the Tools folder in your WDK installation and locate the DevCon tool. For example, look in the following folder:

   C:\Program Files (x86)\Windows Kits\10\Tools\x64\devcon.exe

3. **-> On the target system**

   Install the driver

   The following instructions show you how to install and test the sample driver.

   The INF file required for installing this driver is *TabletAudioSample.inf*. On the target computer, open a Command Prompt window as Administrator. Navigate to your driver package folder, right-click the TabletAudioSample.inf file, and then select **Install**.

   A dialog box will appear indicating that the test driver is an unsigned driver. Click **Install this driver anyway** to proceed.

> **TIP**
> If you have any issues with the installation, check the following file for more information.
> `%windir%\inf\setupapi.dev.log`

For more detailed instructions, see Configuring a Computer for Driver Deployment, Testing, and Debugging.

The INF file contains the hardware ID for installing the *tabletaudiosample.sys*. For the Syvad sample, the hardware ID is: `root\sysvad_TabletAudioSample`

On the target computer, open a Command Prompt window as Administrator. Navigate to your driver package folder, and enter the following command: `devcon status root\sysvad_TabletAudioSample`

Status information is displayed durring the devcon install.

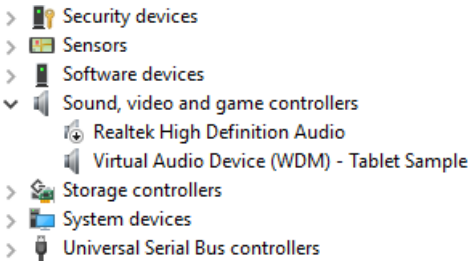
4. **Examine the driver in Device Manager**

   On the target computer, in a Command Prompt window, enter**devmgmt** to open Device Manager. In Device Manager, on the View menu, select **Devices by type**.

   In the device tree, locate *Virtual Audio Device (WDM) - Tablet Sample* in the Audio Device node. This is typically under the **Sound, video and game controllers** node. Confirm that it is installed and active.

   Highlight the driver for the actual hardware on the PC in Device Manager. Then right-click the driver and click disable to disable the driver.

   Confirm in Device Manager that audio hardware driver, displays the a down arrow, indicating that it is disabled.



   After successfully installing the sample driver, you're now ready to test it.

Test the Sysvad audio driver

1. On the target computer, in a Command Prompt window, enter**devmgmt** to open Device Manager. In Device Manager, on the **View** menu, select **Devices by type**. In the device tree, locate *Virtual Audio Device (WDM) - Tablet Sample*.

2. Open Control Panel and navigate to **Hardware and Sound** > **Manage audio devices**. In the Sound dialog box, select the speaker icon labeled as *Virtual Audio Device (WDM) - Tablet Sample*, and then click **Set Default**, but do not click **OK**. This will keep the Sound dialog box open.

3. Locate an MP3 or other audio file on the target computer and double-click to play it. Then in the Sound dialog box, verify that there is activity in the volume level indicator associated with the *Virtual Audio Device (WDM) - Tablet Sample* driver.

Section 5: Use WinDbg to display information about the driver

In Section 5, you will set the symbol path and use kernel debugger commands to display information about the Sysvad sample driver.

Symbols allow for WinDbg to display additional information such as variable names, that can be invaluable when debugging. WinDbg uses the Microsoft Visual Studio debug symbol formats for source-level debugging. It can access any symbol or variable from a module that has PDB symbol files.

To load the debugger, perform the following steps.

<-On the host system

1. If you closed the debugger, open it again using the following command in the administrator command prompt window. Replace the key and port with what you previously configured.

   ```
   C:\> WinDbg –k net:port=50010,key=2steg4fzbj2sz.23418vzkd4ko3.1g34ou07z4pev.1sp3yo9yz874p
   ```

2. Use Ctrl+Break (Scroll Lock) to break into the code running on the target system.

Set the symbol path

1. To set the symbols path to the Microsoft symbol server in the WinDbg environment, use the **.symfix** command.

   ```
   0: kd> .symfix
   ```

2. To add your local symbol location to use your local symbols, add the path using **.sympath+** and then **.reload /f**.

   ```
   0: kd> .sympath+ C:\WDK_Samples\Sysvad
   0: kd> .reload /f
   ```

   Note The **.reload** command with the **/f** force option deletes all symbol information for the specified module and reloads the symbols. In some cases, this command also reloads or unloads the module itself.

Note You must load the proper symbols to use advanced functionality that WinDbg provides. If you do not have symbols properly configured, you will receive messages indicating that symbols are not available when you attempt to use functionality that is dependent on symbols.

```
0:000> dv
Unable to enumerate locals, HRESULT 0x80004005
Private symbols (symbols.pri) are required for locals.
Type “.hh dbgerr005” for details.
```

Note Symbol servers

There are a number of approaches that can be used to work with symbols. In many situations, you can configure the PC to access symbols from a symbol server that Microsoft provides when they are needed. This walkthrough assumes that this approach will be used. If the symbols in your environment are in a different location, modify the steps to use that location. For additional information, see Symbol Stores and Symbol Servers.

Note Understand source code symbol requirements

To perform source debugging, you must build a checked (debug) version of your binaries. The compiler will create symbol files (.pdb files). These symbol files will show the debugger how the binary instructions correspond to the source lines. The actual source files themselves must also be accessible to the debugger.

The symbol files do not contain the text of the source code. For debugging, it is best if the linker does not optimize your code. Source debugging and access to local variables are more difficult, and sometimes nearly impossible, if the code has been optimized. If you are having problems viewing local variables or source lines, set the following

build options.

set COMPILE_DEBUG=1

set ENABLE_OPTIMIZER=0

1. Type the following in the command area of the debugger to display information about the Sysvad driver.

```
0: kd> lm m tabletaudiosample v
Browse full module list
start             end                 module name
fffff801`14b40000 fffff801`14b86000   tabletaudiosample   (private pdb symbols)
C:\Debuggers\sym\TabletAudioSample.pdb\E992C4803EBE48C7B23DC1596495CE181\TabletAudioSample.pdb
    Loaded symbol image file: tabletaudiosample.sys
    Image path: \SystemRoot\system32\drivers\tabletaudiosample.sys
    Image name: tabletaudiosample.sys
    Browse all global symbols  functions  data
    Timestamp:        Thu Dec 10 12:20:26 2015 (5669DE8A)
    CheckSum:         0004891E
...
```

   For more information, see **lm**.

2. Click the **Browse all global symbols** link in the debug output to display information about items symbols that start with the letter a.

3. Because DML is enabled, some elements of the output are hot links that you can click on. Click on the *data* link in the debug output to display information about items symbols that start with the letter a.

```
0: kd> x /D /f tabletaudiosample!a*
 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

fffff806`9adb1000 tabletaudiosample!AddDevice (struct _DRIVER_OBJECT *, struct _DEVICE_OBJECT *)
```

   For information, see **x (Examine Symbols)**.

4. The **!lmi** extension displays detailed information about a module. Type **!lmi tabletaudiosample**. Your output should be similar to the text shown below.

```
0: kd> !lmi tabletaudiosample
Loaded Module Info: [tabletaudiosample]
         Module: tabletaudiosample
   Base Address: fffff8069ad90000
     Image Name: tabletaudiosample.sys
   Machine Type: 34404 (X64)
     Time Stamp: 58ebe848 Mon Apr 10 13:17:12 2017
           Size: 48000
       CheckSum: 42df7
Characteristics: 22
Debug Data Dirs: Type  Size     VA  Pointer
             CODEVIEW    a7,  e5f4,    d1f4 RSDS - GUID: {5395F0C5-AE50-4C56-AD31-DD5473BD318F}
               Age: 1, Pdb: C:\Windows-driver-samples-
master\audio\sysvad\TabletAudioSample\x64\Debug\TabletAudioSample.pdb
                   ??   250,  e69c,    d29c [Data not mapped]
     Image Type: MEMORY   - Image read successfully from loaded memory.
    Symbol Type: PDB      - Symbols loaded successfully from image header.
                 C:\Program Files (x86)\Windows
Kits\10\Debuggers\x64\sym\TabletAudioSample.pdb\5395F0C5AE504C56AD31DD5473BD318F1\TabletAudioSample.pdb
       Compiler: Resource - front end [0.0 bld 0] - back end [14.0 bld 24210]
    Load Report: private symbols & lines, not source indexed
                 C:\Program Files (x86)\Windows
Kits\10\Debuggers\x64\sym\TabletAudioSample.pdb\5395F0C5AE504C56AD31DD5473BD318F1\TabletAudioSample.pdb
```

5. Use the **!dh** extension to display header information as shown below.

```
0: kd> !dh tabletaudiosample

File Type: EXECUTABLE IMAGE
FILE HEADER VALUES
    8664 machine (X64)
       9 number of sections
5669DE8A time date stamp Thu Dec 10 12:20:26 2015

       0 file pointer to symbol table
       0 number of symbols
      F0 size of optional header
      22 characteristics
            Executable
            App can handle >2gb addresses
...
```

Section 6: Displaying Plug and Play device tree information

In Section 6, you will display information about the Sysvad sample device driver and where it lives in the Plug and Play device tree.

Information about the device driver in the Plug and Play device tree can be useful for troubleshooting. For example, if a device driver is not resident in the device tree, there may an issue with the installation of the device driver.

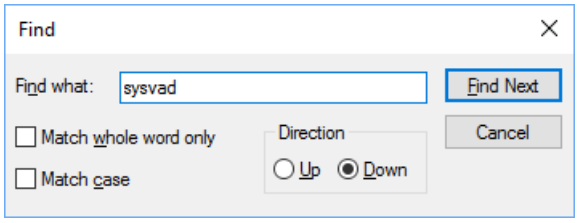
For more information about the device node debug extension, see **!devnode**.

<-On the host system

1. To see all the device nodes in the Plug and Play device tree, enter the **!devnode 0 1** command. This command can take a minute or two to run. During that time, "*Busy" will be displayed in the status area of WinDbg.

```
0: kd> !devnode 0 1
Dumping IopRootDeviceNode (= 0xffffe0005a3a8d30)
DevNode 0xffffe0005a3a8d30 for PDO 0xffffe0005a3a9e50
  InstancePath is "HTREE\ROOT\0"
  State = DeviceNodeStarted (0x308)
  Previous State = DeviceNodeEnumerateCompletion (0x30d)
  DevNode 0xffffe0005a3a3d30 for PDO 0xffffe0005a3a4e50
    InstancePath is "ROOT\volmgr\0000"
    ServiceName is "volmgr"
    State = DeviceNodeStarted (0x308)
    Previous State = DeviceNodeEnumerateCompletion (0x30d)
    DevNode 0xffffe0005a324560 for PDO 0xffffe0005bd95ca0…
...
```

2. Use Ctrl+F to search in the output that is generated to look for the name of the device driver, *sysvad*.



A device node entry with a name of `sysvad_TabletAudioSample` will be present in the !devnode output for Syvad.

```
  DevNode 0xffffe00086e68190 for PDO 0xffffe00089c575a0
    InstancePath is "ROOT\sysvad_TabletAudioSample\0000"
    ServiceName is "sysvad_tabletaudiosample"
    State = DeviceNodeStarted (0x308)
...
```

Note that the PDO address and the DevNode address are displayed.

3. Use the `!devnode 0 1 sysvad_TabletAudioSample` command to display Plug and Play information associated with our Sysvad device driver.

```
0: kd> !devnode 0 1 sysvad_TabletAudioSample
Dumping IopRootDeviceNode (= 0xffffe00082df8d30)
DevNode 0xffffe00086e68190 for PDO 0xffffe00089c575a0
  InstancePath is "ROOT\sysvad_TabletAudioSample\0000"
  ServiceName is "sysvad_tabletaudiosample"
  State = DeviceNodeStarted (0x308)
  Previous State = DeviceNodeEnumerateCompletion (0x30d)
  DevNode 0xffffe000897fb650 for PDO 0xffffe00089927e30
    InstancePath is "SWD\MMDEVAPI\{0.0.0.00000000}.{64097438-cdc0-4007-a19e-62e789062e20}"
    State = DeviceNodeStarted (0x308)
    Previous State = DeviceNodeStartPostWork (0x307)
  DevNode 0xffffe00086d2f5f0 for PDO 0xffffe00089939ae0
    InstancePath is "SWD\MMDEVAPI\{0.0.0.00000000}.{78880f4e-9571-44a4-a9df-960bde446487}"
    State = DeviceNodeStarted (0x308)
    Previous State = DeviceNodeStartPostWork (0x307)
  DevNode 0xffffe00089759bb0 for PDO 0xffffe000875aa060
    InstancePath is "SWD\MMDEVAPI\{0.0.0.00000000}.{7cad07f2-d0a0-4b9b-8100-8dc735e9c447}"
    State = DeviceNodeStarted (0x308)
    Previous State = DeviceNodeStartPostWork (0x307)
  DevNode 0xffffe00087735010 for PDO 0xffffe000872068c0
    InstancePath is "SWD\MMDEVAPI\{0.0.0.00000000}.{fc38551b-e69f-4b86-9661-ae6da78bc3c6}"
    State = DeviceNodeStarted (0x308)
    Previous State = DeviceNodeStartPostWork (0x307)
  DevNode 0xffffe00088457670 for PDO 0xffffe0008562b830
    InstancePath is "SWD\MMDEVAPI\{0.0.1.00000000}.{0894b831-c9fe-4c56-86a6-092380fc5628}"
    State = DeviceNodeStarted (0x308)
    Previous State = DeviceNodeStartPostWork (0x307)
  DevNode 0xffffe000893dbb70 for PDO 0xffffe00089d68060
    InstancePath is "SWD\MMDEVAPI\{0.0.1.00000000}.{15eb6b5c-aa54-47b8-959a-0cff2c1500db}"
    State = DeviceNodeStarted (0x308)
    Previous State = DeviceNodeStartPostWork (0x307)
  DevNode 0xffffe00088e6f250 for PDO 0xffffe00089f6e990
    InstancePath is "SWD\MMDEVAPI\{0.0.1.00000000}.{778c07f0-af9f-43f2-8b8d-490024f87239}"
    State = DeviceNodeStarted (0x308)
    Previous State = DeviceNodeStartPostWork (0x307)
  DevNode 0xffffe000862eb4b0 for PDO 0xffffe000884443a0
    InstancePath is "SWD\MMDEVAPI\{0.0.1.00000000}.{e4b72c7c-be50-45df-94f5-0f2922b85983}"
    State = DeviceNodeStarted (0x308)
    Previous State = DeviceNodeStartPostWork (0x307)
```

4. The output displayed in the previous command includes the PDO associated with the running instance of our driver, in this example it is *0xffffe00089c575a0*. Enter the **!devobj**<*PDO address*> command to display Plug and Play information associated with the Sysvad device driver. Use the PDO address that **!devnode** displays on your PC, not the one shown here.

```
0: kd> !devobj 0xffffe00089c575a0
Device object (ffffe00089c575a0) is for:
0000004e \Driver\PnpManager DriverObject ffffe00082d47e60
Current Irp 00000000 RefCount 65 Type 0000001d Flags 00001040
SecurityDescriptor ffffc102b0f6d171 DevExt 00000000 DevObjExt ffffe00089c576f0 DevNode ffffe00086e68190
ExtensionFlags (0000000000)
Characteristics (0x00000180)  FILE_AUTOGENERATED_DEVICE_NAME, FILE_DEVICE_SECURE_OPEN
AttachedDevice (Upper) ffffe00088386a50 \Driver\sysvad_tabletaudiosample
Device queue is not busy.
```

5. The output displayed in the **!devobj** command includes the name of the attached device: \Driver\sysvad_tabletaudiosample. Use the **!drvobj** command with a bit mask of 2, to display information associated with the attached device.

```
0: kd> !drvobj \Driver\sysvad_tabletaudiosample 2
Driver object (ffffe0008834f670) is for:
\Driver\sysvad_tabletaudiosample
DriverEntry:   fffff80114b45310  tabletaudiosample!FxDriverEntry
DriverStartIo: 00000000
DriverUnload:  fffff80114b5fea0               tabletaudiosample!DriverUnload
AddDevice:     fffff80114b5f000  tabletaudiosample!AddDevice

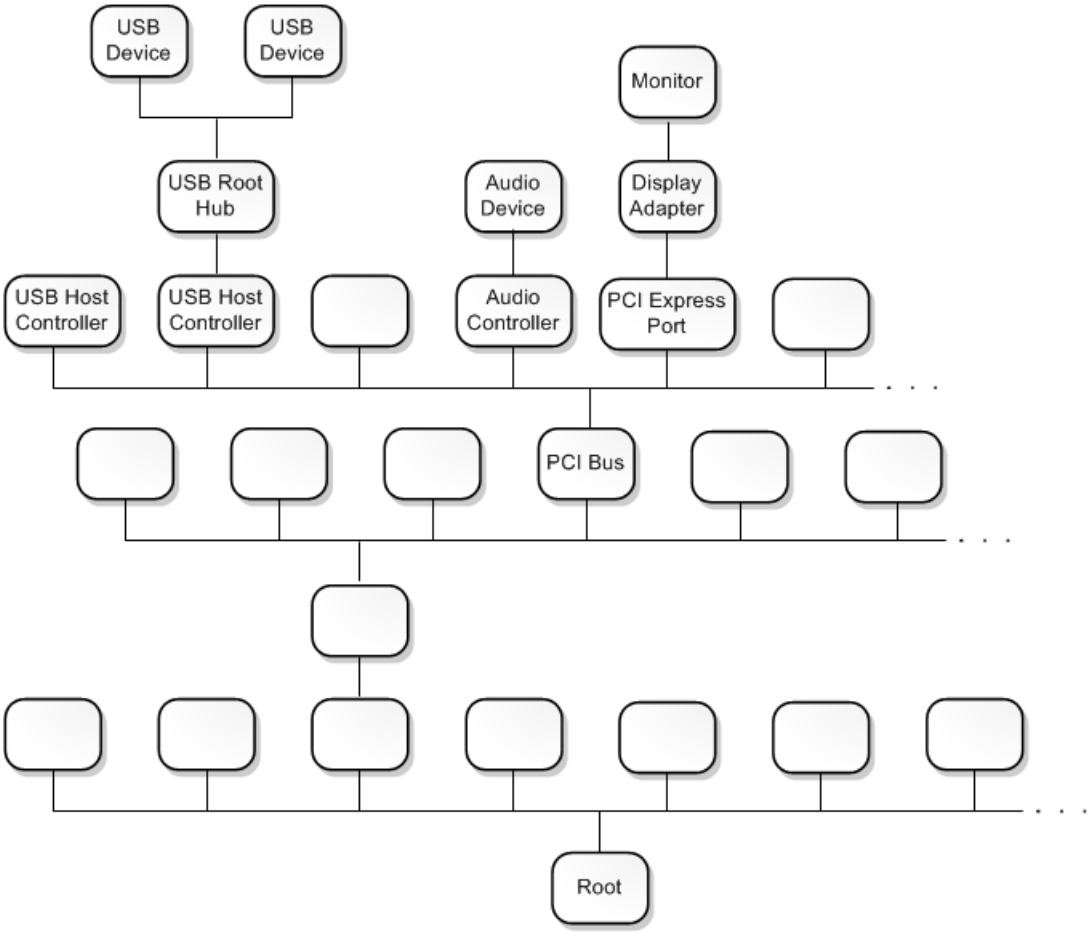
Dispatch routines:
[00] IRP_MJ_CREATE                      fffff80117b49a20             portcls!DispatchCreate
[01] IRP_MJ_CREATE_NAMED_PIPE           fffff8015a949a00         nt!IopInvalidDeviceRequest
[02] IRP_MJ_CLOSE                       fffff80115e26f90               ks!DispatchCleanup
[03] IRP_MJ_READ                        fffff80115e32710               ks!DispatchRead
[04] IRP_MJ_WRITE                       fffff80115e327e0             ks!DispatchWrite
[05] IRP_MJ_QUERY_INFORMATION           fffff8015a949a00        nt!IopInvalidDeviceRequest
[06] IRP_MJ_SET_INFORMATION             fffff8015a949a00             nt!IopInvalidDeviceRequest
[07] IRP_MJ_QUERY_EA                    fffff8015a949a00        nt!IopInvalidDeviceRequest
[08] IRP_MJ_SET_EA                      fffff8015a949a00             nt!IopInvalidDeviceRequest
[09] IRP_MJ_FLUSH_BUFFERS               fffff80115e32640  ks!DispatchFlush
[0a] IRP_MJ_QUERY_VOLUME_INFORMATION    fffff8015a949a00          nt!IopInvalidDeviceRequest
[0b] IRP_MJ_SET_VOLUME_INFORMATION      fffff8015a949a00              nt!IopInvalidDeviceRequest
[0c] IRP_MJ_DIRECTORY_CONTROL           fffff8015a949a00          nt!IopInvalidDeviceRequest
[0d] IRP_MJ_FILE_SYSTEM_CONTROL         fffff8015a949a00        nt!IopInvalidDeviceRequest
[0e] IRP_MJ_DEVICE_CONTROL              fffff80115e27480              ks!DispatchDeviceIoControl
[0f] IRP_MJ_INTERNAL_DEVICE_CONTROL     fffff8015a949a00   nt!IopInvalidDeviceRequest
[10] IRP_MJ_SHUTDOWN                    fffff8015a949a00      nt!IopInvalidDeviceRequest
[11] IRP_MJ_LOCK_CONTROL                fffff8015a949a00  nt!IopInvalidDeviceRequest
[12] IRP_MJ_CLEANUP                     fffff8015a949a00          nt!IopInvalidDeviceRequest
[13] IRP_MJ_CREATE_MAILSLOT             fffff8015a949a00              nt!IopInvalidDeviceRequest
[14] IRP_MJ_QUERY_SECURITY              fffff80115e326a0 ks!DispatchQuerySecurity
[15] IRP_MJ_SET_SECURITY                fffff80115e32770      ks!DispatchSetSecurity
[16] IRP_MJ_POWER                       fffff80117b3dce0            portcls!DispatchPower
[17] IRP_MJ_SYSTEM_CONTROL              fffff80117b13d30              portcls!PcWmiSystemControl
[18] IRP_MJ_DEVICE_CHANGE               fffff8015a949a00 nt!IopInvalidDeviceRequest
[19] IRP_MJ_QUERY_QUOTA                 fffff8015a949a00  nt!IopInvalidDeviceRequest
[1a] IRP_MJ_SET_QUOTA                   fffff8015a949a00       nt!IopInvalidDeviceRequest
[1b] IRP_MJ_PNP                         fffff80114b5f7d0 tabletaudiosample!PnpHandler
```

6. Enter the **!devstack**<*PDO address*> command to display Plug and Play information associated with the device driver. The output displayed in the **!devnode 0 1** command includes the PDO address associated with the running instance of our driver. In this example it is *0xffffe00089c575a0*. Use the PDO address that **!devnode** displays on your PC, not the one shown below.

```
0: kd> !devstack 0xffffe00089c575a0
  !DevObj           !DrvObj            !DevExt           ObjectName
  ffffe00088d212e0  \Driver\ksthunk    ffffe00088d21430  0000007b
  ffffe00088386a50  \Driver\sysvad_tabletaudiosampleffffe00088386ba0  0000007a
> ffffe00089c575a0  \Driver\PnpManager 00000000  0000004e
!DevNode ffffe00086e68190 :
  DeviceInst is "ROOT\sysvad_TabletAudioSample\0000"
  ServiceName is "sysvad_tabletaudiosample"
```

The output shows that we have a farily simple device driver stack. The sysvad_TabletAudioSample driver is a child of the PnPManager node. The PnPManager is a root node.

This diagram shows a more complex device node tree.



Note For more information about more complex driver stacks, see Driver stacks and Device nodes and device stacks.

Section 7: Working with breakpoints

In Section 7, you will work with breakpoints to stop code execution at specific points.

Setting breakpoints using commands

Breakpoints are used to stop code execution at a particular line of code. You can then step forward in the code from that point, to debug that specific section of code.

To set a breakpoint using a debug command, use one of the following **b** commands.

bp	Sets a breakpoint that will be active until the module it is in is unloaded.
bu	Sets a breakpoint that is unresolved when the module is unloaded and re-enables when the module reloads.
bm	Sets a breakpoint for a symbol. This command will use bu or bp appropriately and allows wildcards * to be used to set breakpoints on every symbols that matches (like all methods in a class).

1. Use the WinDbg UI to confirm that **Debug** > **Source Mode** is enabled in the current WinDbg session.

2. Add your local code location to the source path by typing the following command.

```
.sympath+ C:\WDK_Samples\Sysvad
```

3. Add your local symbol location to the symbol path by typing the following command.

```
.sympath+ C:\WDK_Samples\Sysvad
```

4. **Set the debug mask**

   As you are working with a driver it can be handy to see all of the messages that it may display. Type the following to change the default debug bit mask so that all debug messages from the target system will be displayed in the debugger.

```
0: kd> ed nt!Kd_DEFAULT_MASK 0xFFFFFFFF
```

5. Set the breakpoint with the **bm** command using the name of the driver, followed by the function name (AddDevice) where you want to set the breakpoint, separated by an exclamation mark.

```
0: kd> bm tabletaudiosample!AddDevice
breakpoint 1 redefined
  1: fffff801`14b5f000 @!"tabletaudiosample!AddDevice"
```

   You can use different syntax in conjunction with setting variables like <module>!<symbol>, <class>::<method>,'<file.cpp>:<line number>', or skip a number of times <condition> <#>. For more information, see Using Breakpoints.

6. List the current breakpoints to confirm that the breakpoint was set by typing the **bl** command.

```
0: kd> bl
1 e fffff801`14b5f000     0001 (0001) tabletaudiosample!AddDevice
```

7. Restart code execution on the target system by typing the go command **g**.

8. **->On the target system**

   In Windows, open Device Manager by using the icon or by entering **mmc devmgmt.msc**. In **Device Manager** expand the **Sound, video and game controllers** node. Right click the virtual audio driver entry and select **Disable** from the menu.

9. Right click the virtual audio driver entry again and select **Enable** from the menu.

10. **<- On the host system**

    This should cause Windows to reload the driver, which calls AddDevice. This will cause the AddDevice debug breakpoint to fire and the execution of the driver code on the target system should halt.

```
Breakpoint 1 hit
tabletaudiosample!AddDevice:
fffff801`14baf000 4889542410      mov     qword ptr [rsp+10h],rdx
```

    If your source path is set properly, you should stop at the AddDevice routine in adapter.cpp

```
{
    PAGED_CODE();

    NTSTATUS        ntStatus;
    ULONG           maxObjects;

    DPF(D_TERSE, ("[AddDevice]"));

    maxObjects = g_MaxMiniports;

    #ifdef SYSVAD_BTH_BYPASS
    //
    // Allow three (3) Bluetooth hands-free profile devices.
    //
    maxObjects += g_MaxBthHfpMiniports * 3;
    #endif // SYSVAD_BTH_BYPASS

    // Tell the class driver to add the device.
    //
    ntStatus =
        PcAddAdapterDevice
        (
            DriverObject,
            PhysicalDeviceObject,
            PCPFNSTARTDEVICE(StartDevice),
            maxObjects,
            0
        );
    return ntStatus;
} // AddDevice
```

11. Step line-by-line through the code by typing the **p** command or pressing F10. You can step forward out of the sysvad AddDevice code to PpvUtilCall, PnpCallAddDevice and then to the PipCallDriverAddDevice Windows code. You can provide a number to the **p** command to step forward multiple lines, for example *p 5*.

12. When you are done stepping through the code, use the go command **g** to restart execution on the target system.

Setting memory access breakpoints

You can also set breakpoints that fire when a memory location is accessed. Use the **ba** (break on access) command, with the following syntax.

```
ba <access> <size> <address> {options}
```

OPTION	DESCRIPTION
e	execute (when CPU fetches an instruction from the address)
r	read/write (when CPU reads or writes to the address)
w	write (when the CPU writes to the address)

Note that you can only set four data breakpoints at any given time and it is up to you to make sure that you are aligning your data correctly or you won't trigger the breakpoint (words must end in addresses divisible by 2,

dwords must be divisible by 4, and quadwords by 0 or 8)

For example, to set a read/write breakpoint on a specific memory address, use a command like this.

```
ba r 4 fffff800`7bc9eff0
```

Modifying breakpoint state

You can modify existing breakpoints by using the following commands.

bl	Lists breakpoints.
bc	Clears a breakpoint from the list. Use bc * to clear all breakpoints.
bd	Disables a breakpoint. Use bd * to disable all breakpoints.
be	Enables a breakpoint. Use be * to enable all breakpoints.

Alternatively, you can also modify breakpoints by clicking **edit** > **breakpoints**. Note that the breakpoint dialog box only works with existing breakpoints. New breakpoints must be set from the command line.

Set a breakpoint on MixerVolume

Different parts of the audio driver code is called to respond to various events, after the device driver is loaded. In the next section, we set a breakpoint that will fire when the user adjusts the volume control for the virtual audio driver.

To set a breakpoint on MixerVolume, perform the following steps.

1. **<- On the host system**

   To locate the method that changes the volume, use the x command to list the symbols in CAdapterCommon, that contain the string volume.

   ```
   kd> x tabletaudiosample!CAdapterCommon::*
   ...
   fffff800`7bce26a0 tabletaudiosample!CAdapterCommon::MixerVolumeWrite (unsigned long, unsigned long,
   long)
   …
   ```

   Use CTRL+F to search upward in the output for volume and locate the MixerVolumeWrite method.

2. Clear the previous breakpoints using bc *.

3. Set a symbol breakpoint on the CAdapterCommon::MixerVolumeWrite routine using the following command.

   ```
   kd> bm tabletaudiosample!CAdapterCommon::MixerVolumeWrite
     1: fffff801`177b26a0 @!"tabletaudiosample!CAdapterCommon::MixerVolumeWrite"
   ```

4. List the breakpoints to confirm that the breakpoint is set properly.

```
kd> bl
1 e fffff801`177b26a0 [c:\WDK_Samples\audio\sysvad\common.cpp @ 1668]    0001 (0001)
tabletaudiosample!CAdapterCommon::MixerVolumeWrite
```

5. Restart code execution on the target system by typing the go command **g**.
6. In Control Panel select **Hardware and Sound** >**Sound**. Right click **Sink Description Sample** and select **Properties**. Select the **Levels** tab. Adjust the slider volume.
7. This should cause the SetMixerVolume debug breakpoint to fire and execution of the driver code on the target system should halt.

   ```
   kd> g
   Breakpoint 1 hit
   tabletaudiosample!CAdapterCommon::MixerVolumeWrite:
   fffff801`177b26a0 44894c2420      mov     dword ptr [rsp+20h],r9d
   ```

   You should stop at this line in common.cpp

   ```
   {
       if (m_pHW)
       {
           m_pHW->SetMixerVolume(Index, Channel, Value);
       }
   } // MixerVolumeWrite
   ```

8. Use the dv command to display the current variables and their values. More information on variables is provided in the next section of this lab.

   ```
   2: kd> dv
              this = 0x00000000`00000010
            ulNode = 0x344
         ulChannel = 0x210a45f8
           lVolume = 0n24
   ```

9. Press **F10** to single step through the code.
10. Press **F5** to finish the execution of the MixerVolumeWrite code.

Summary - Stepping through code from the Debugger Command window

The following are the commands that you can use to step through your code (with the associated keyboard short cuts shown in parentheses).

- Break in (Ctrl+Break) - This command will interrupt a system as long as the system is running and is in communication with WinDbg (the sequence in the Kernel Debugger is Ctrl+C).
- Step over (F10) – This command causes code execution to proceed one statement or one instruction at a time. If a call is encountered, code execution passes over the call without entering the called routine. (If the programming language is C or C++ and WinDbg is in source mode, source mode can be turned on or off using **Debug**>**Source Mode**).
- Step in (F11) – This command is like step-over, except that the execution of a call does go into the called routine.
- Step out (Shift+F11) – This command causes execution to run to and exit from the current routine (current place in the call stack). This is useful if you've seen enough of the routine.

- Run to cursor (F7 or Ctrl+F10) – Place the cursor in a source or disassembly window where you want the execution to break, then press F7; code execution will run to that point. Note that if the flow of code execution does not reach the point indicated by the cursor (e.g., an IF statement isn't executed), WinDbg would not break, because the code execution did not reach the indicated point.
- Run (F5) – Run until a breakpoint is encountered or an event like a bug check occurs.

Advanced options

- Set instruction to the current line (Ctrl+Shift+I) – In a source window, you can place your cursor on a line, enter this keyboard shortcut, and code execution will start from that point as soon as you let it proceed (for example using F5 or F10). This is handy if you want to retry a sequence, but it requires some care. For example, registers and variables are not set to what they would be if code execution had reached that line naturally.
- Direct setting of the eip register -- You can put a value into the eip register, and as soon as you press F5 (or F10, F11, etc.), execution commences from that address. This is similar to setting instruction to the cursor-designated current line, except that you specify the address of an assembly instruction.

It can be easier to step through UI rather than from the command line so this method is recommended. If necessary, the following commands can be used to step through a source file at the command line:

- .lines - Enable source line information.
- bp main - Set the initial breakpoint at the beginning of your module.
- l+t - Stepping will be done by source line.
- Select **Debug**>**Source Mode** to enter source mode; the `L+t` command is not sufficient.
- l+s - Source lines will be displayed at prompt.
- g - Run program until "main" is entered.
- p - Execute one source line.

For more information, see Source Code Debugging in WinDbg in the debugging reference documentation.

Set breakpoints in code

You can set a breakpoint in code by adding the `DebugBreak()` statement and rebuilding the project and re-installing the driver. This breakpoint will fire each time the driver is enabled, so it would be a techniques to be used in the early development stages, not in production code. This technique is not as flexible as dynamically setting breakpoints using the breakpoint commands.

Tip: You may want to keep a copy of the Sysvad driver with out the breakpoint added for further lab work.

1. Set a break to occur each time the AddDevice method is run by adding the `DebugBreak()` statement to the sample code.

```
...
    // Insert the DebugBreak() statment before the  PcAddAdapterDevice is called.
    //

    DebugBreak()

    // Tell the class driver to add the device.
    //
    ntStatus =
        PcAddAdapterDevice
        (
            DriverObject,
            PhysicalDeviceObject,
            PCPFNSTARTDEVICE(StartDevice),
            maxObjects,
            0
        );

    return ntStatus;
} // AddDevice
```

2. Follow all of the steps previously described to rebuild the driver in Microsoft Visual Studio and re-install it to the target machine. Be sure to uninstall the existing driver before installing the updated driver.
3. Clear any previous breakpoints and make sure that the debugger is attached to the target PC.
4. When the code runs and reaches the `DebugBreak` statement, execution will stop and a message will be displayed.

```
KERNELBASE!DebugBreak:
77b3b770 defe     __debugbreak
```

Section 8: Display variables

In Section 8, you will use debugger commands to display variables.

It can be useful to examine variables as the code executes to confirm that the code is working as expected. This labs examines variables as the audio driver produces sound.

1. Use the **dv** command to examine the locale variables associated with the tabletaudiosample!CMiniportWaveRT::New*.

```
kd> dv tabletaudiosample!CMiniportWaveRT::New*
```

2. Clear the previous breakpoints

```
bc *
```

3. Set a symbol breakpoint on the CMiniportWaveCyclicStreamMSVAD routines using the following command.

```
0: kd> bm tabletaudiosample!CMiniportWaveRT::NewStream
  1: fffff801`177dffc0 @!"tabletaudiosample!CMiniportWaveRT::NewStream"
```

4. Restart code execution on the target system by typing the go command **g**.

5. **-> On the target system**

   Locate a small media file (such as Windows notification sound file with a .wav file extension) and click the file to play it. For example you can use Ring05.wav located in the Windows\Media directory.

6. **<- On the host system**

   When the media file is played, the breakpoint should fire, and execution of the driver code on the target system should halt.

```
Breakpoint 1 hit
tabletaudiosample!CMiniportWaveRT::NewStream:
fffff801`177dffc0 44894c2420      mov     dword ptr [rsp+20h],r9d
```

   The source code Window should be highlighting the brace on the entrance to the NewStream function.

```
/*++

Routine Description:

  The NewStream function creates a new instance of a logical stream
  associated with a specified physical channel. Callers of NewStream should
  run at IRQL PASSIVE_LEVEL.

Arguments:

  OutStream -

  OuterUnknown -

  Pin -

  Capture -

  DataFormat -

Return Value:

  NT status code.

--*/
{

...
```

7. **Local variables**

   You can display the names and values of all local variables for a given frame by typing the **dv** command.

```
0: kd> dv
                this = 0xffffe000`4436f8e0
           OutStream = 0xffffe000`49d2f130
        OuterUnknown = 0xffffe000`4436fa30
                 Pin = 0
             Capture = 0x01 '
          DataFormat = 0xffffe000`44227790
signalProcessingMode = {487E9220-E000-FFFF-30F1-D24900E0FFFF}
            ntStatus = 0n1055
              stream = 0x00000000`00000200
```

8. **Use DML to Display Variables**

To use DML to explore variables, click the underlined elements. The click action builds a **dx (Display NatVis Expression)** command that allows you to drill down on nested data structures.

```
0: kd> dx -r1 (*((tabletaudiosample!CMiniportWaveRT *)0xffffe001d10b8380))
(*((tabletaudiosample!CMiniportWaveRT *)0xffffe001d10b8380)) :  [Type: CMiniportWaveRT]
    [+0x020] m_lRefCount      : 0
    [+0x028] m_pUnknownOuter  : 0xffffe001d1477e50 : [Type: IUnknown *]
    [+0x030] m_ulLoopbackAllocated : 0x2050
    [+0x034] m_ulSystemAllocated : 0x180
    [+0x038] m_ulOffloadAllocated : 0x0
    [+0x03c] m_dwCaptureAllocatedModes : 0x0

0: kd> dx -r1 (*((tabletaudiosample!_GUID *)0xffffd001c8acd348))
(*((tabletaudiosample!_GUID *)0xffffd001c8acd348)) : {487E9220-E000-FFFF-30F1-D24900E0FFFF} [Type:
_GUID]
    [<Raw View>]

0: kd> dx -r1 -n (*((tabletaudiosample!_GUID *)0xffffd001c8acd348))
(*((tabletaudiosample!_GUID *)0xffffd001c8acd348)) :  [Type: _GUID]
    [+0x000] Data1            : 0x487e9220
    [+0x004] Data2            : 0xe000
    [+0x006] Data3            : 0xffff
    [+0x008] Data4            :  [Type: unsigned char [8]]

0: kd> dx -r1 -n (*((tabletaudiosample!unsigned char (*)[8])0xffffd001c8acd350))
(*((tabletaudiosample!unsigned char (*)[8])0xffffd001c8acd350)) :  [Type: unsigned char [8]]
    [0]              : 0x30
    [1]              : 0xf1
    [2]              : 0xd2
    [3]              : 0x49
    [4]              : 0x0
    [5]              : 0xe0
    [6]              : 0xff
    [7]              : 0xff
```

9. **Global variables**

   You can find the memory location of a global variable by typing **? <variable name>**.

   ```
   0: kd> ? signalProcessingMode
   Evaluate expression: -52768896396472 = ffffd001`c8acd348
   ```

10. This returns the memory location of the variable, in this case *ffffd001`c8acd348*. You can view the contents of the memory location by dumping the value of that location typing the **dd** command using the memory location returned by the previous command.

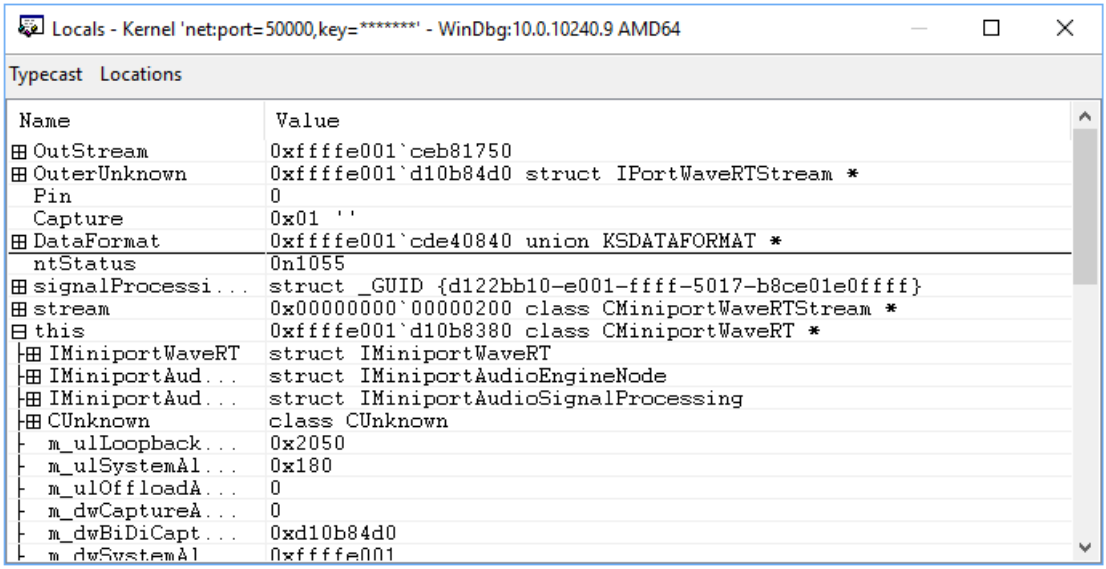
    ```
    0: kd> dd ffffd001`c8acd348
    ffffd001`c8acd348  487e9220 ffffe000 49d2f130 ffffe000
    ffffd001`c8acd358  4837c468 ffffe000 18221570 ffffc000
    ffffd001`c8acd368  4436f8e0 ffffe000 487e9220 ffffe000
    ffffd001`c8acd378  18ab145b fffff801 4837c420 ffffe000
    ffffd001`c8acd388  4436f8e0 ffffe000 49d2f130 ffffe000
    ffffd001`c8acd398  4436fa30 ffffe000 00000000 00000000
    ffffd001`c8acd3a8  00000001 00000000 44227790 ffffe000
    ffffd001`c8acd3b8  18adc7f9 fffff801 495972a0 ffffe000
    ```

11. You can also use variable names with the **dd** command.

```
0: kd> dd signalProcessingMode
ffffd001`c8acd348  487e9220 ffffe000 49d2f130 ffffe000
ffffd001`c8acd358  4837c468 ffffe000 18221570 ffffc000
ffffd001`c8acd368  4436f8e0 ffffe000 487e9220 ffffe000
ffffd001`c8acd378  18ab145b fffff801 4837c420 ffffe000
ffffd001`c8acd388  4436f8e0 ffffe000 49d2f130 ffffe000
ffffd001`c8acd398  4436fa30 ffffe000 00000000 00000000
ffffd001`c8acd3a8  00000001 00000000 44227790 ffffe000
ffffd001`c8acd3b8  18adc7f9 fffff801 495972a0 ffffe000
```

12. **Display variables**

    Use the **View**> **Locals** menu item to display local variables. This interface also provides this ability to drill down on more complex data structures.



13. Use p or F10 to step forward about 10 lines in the code until you are highlighting the ntStatus = IsFormatSupported(Pin, Capture, DataFormat); line of code.

```
    PAGED_CODE();

    ASSERT(OutStream);
    ASSERT(DataFormat);

    DPF_ENTER(("[CMiniportWaveRT::NewStream]"));

    NTSTATUS                    ntStatus = STATUS_SUCCESS;
    PCMiniportWaveRTStream      stream = NULL;
    GUID                        signalProcessingMode = AUDIO_SIGNALPROCESSINGMODE_DEFAULT;

    *OutStream = NULL;

     //
    // If the data format attributes were specified, extract them.
    //
    if ( DataFormat->Flags & KSDATAFORMAT_ATTRIBUTES )
    {
        // The attributes are aligned (QWORD alignment) after the data format
        PKSMULTIPLE_ITEM attributes = (PKSMULTIPLE_ITEM) (((PBYTE)DataFormat) + ((DataFormat-
>FormatSize + FILE_QUAD_ALIGNMENT) & ~FILE_QUAD_ALIGNMENT));
        ntStatus = GetAttributesFromAttributeList(attributes, attributes->Size, &signalProcessingMode);
    }

    // Check if we have enough streams.
    //
    if (NT_SUCCESS(ntStatus))
    {
        ntStatus = ValidateStreamCreate(Pin, Capture, signalProcessingMode);
    }

    // Determine if the format is valid.
    //
    if (NT_SUCCESS(ntStatus))
    {
        ntStatus = IsFormatSupported(Pin, Capture, DataFormat);
    }

...
```

14. Use the **dv** command to display the names and values of all local variables for a given frame. Note that, as expected, the values are different from the last time we ran this command, as additional code has been run that changes the local variables and some variables are now not in the current frame or their values have changed.

```
2: kd> dv
                this = 0xffffe001`d1182000
           OutStream = 0xffffe001`d4776d20
        OuterUnknown = 0xffffe001`d4776bc8
                 Pin = 0
             Capture = 0x00 '
          DataFormat = 0xffffe001`cd7609b0
signalProcessingMode = {4780004E-7133-41D8-8C74-660DADD2C0EE}
           ntStatus = 0n0
             stream = 0x00000000`00000000
```

Section 9: View call stacks

In Section 9, you will view call stacks to examine caller/calle code.

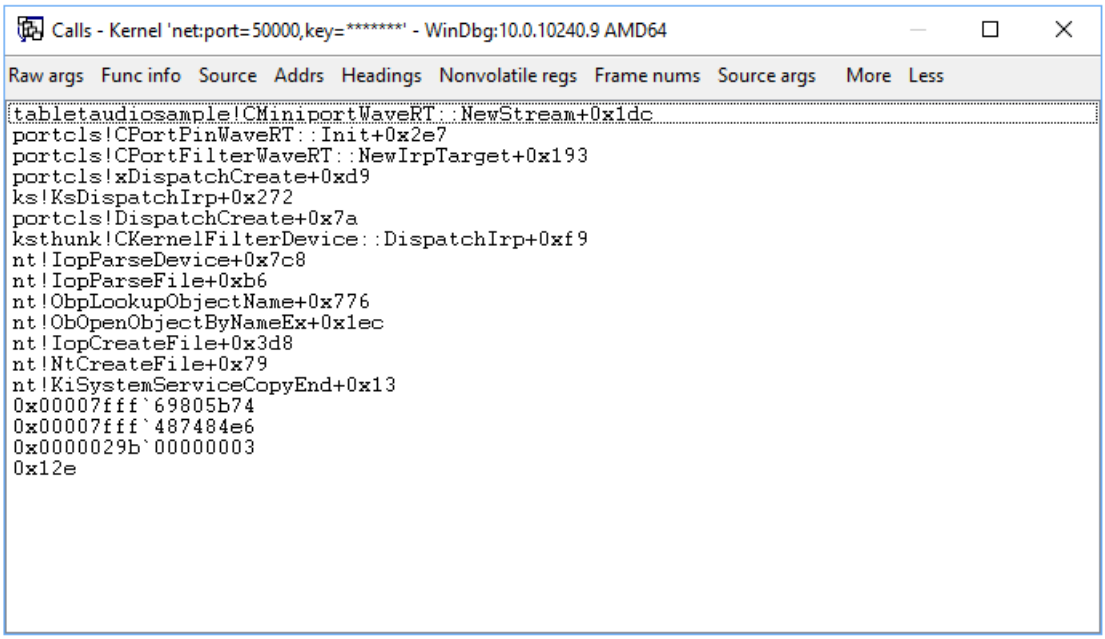
The call stack is the chain of function calls that have led to the current location of the program counter. The top function on the call stack is the current function, and the next function is the function that called the current

function, and so on.

To display the call stack, use the k* commands:

kb	Displays the stack and first three parameters.
kp	Displays the stacks and the full list of parameters.
kn	Allows you to see the stack with the frame information next to it.

If you want to keep the call stack available, you can click **View**> **Call stack** to view it. Click the columns at the top of the window to toggle the display of additional information.



This output shows the call stack while debugging the sample adapter code in a break state.

```
0: kd> kb
# RetAddr           : Args to Child                                                   : Call Site
00 fffff800`7a0fa607 : ffffe001`d1182000 ffffe001`d4776d20 ffffe001`d4776bc8 ffffe001`00000000 :
tabletaudiosample!CMiniportWaveRT::NewStream+0x1dc [c:\data1\threshold\audio\endpointscommon\minwavert.cpp @
597]
01 fffff800`7a0fb2c3 : 00000000`00000000 ffffe001`d122bb10 ffffe001`ceb81750 ffffe001`d173f058 :
portcls!CPortPinWaveRT::Init+0x2e7
02 fffff800`7a0fc7f9 : ffffe001`d4776bc0 00000000`00000000 ffffe001`d10b8380 ffffe001`d122bb10 :
portcls!CPortFilterWaveRT::NewIrpTarget+0x193
03 fffff800`7a180552 : 00000000`00000000 ffffe001`d10b8380 ffffe001`d122bb10 ffffe001`d4565600 :
portcls!xDispatchCreate+0xd9
04 fffff800`7a109a9a : ffffe001`d10b84d0 ffffe001`d10b8380 00000000`00000000 ffffe001`00000000 :
ks!KsDispatchIrp+0x272
05 fffff800`7bd314b1 : ffffe001`d122bb10 ffffd001`c3098590 ffffe001`d122bd90 ffffe001`ce80da70 :
portcls!DispatchCreate+0x7a
06 fffff803`cda1bfa8 : 00000000`00000024 00000000`00000000 00000000`00000000 ffffe001`d122bb10 :
ksthunk!CKernelFilterDevice::DispatchIrp+0xf9
07 fffff803`cda7b306 : 00000000`000001f0 ffffe001`d48ce690 ffffe001`d13d6400 ffffe001`d13d64c0 :
nt!IopParseDevice+0x7c8
08 fffff803`cda12916 : 00000000`000001f0 ffffd001`c30988d0 ffffe001`d13d6490 fffff803`cda7b250 :
nt!IopParseFile+0xb6
09 fffff803`cda1131c : ffffe001`d2ccb001 ffffd001`c30989e0 00ffffe0`00000040 ffffe001`cd127dc0 :
nt!ObpLookupObjectName+0x776
0a fffff803`cd9fedb8 : ffffe001`00000001 ffffe001`d48ce690 00000000`00000000 00000000`00000000 :
nt!ObOpenObjectByNameEx+0x1ec
0b fffff803`cd9fe919 : 000000ee`6d1fc8d8 000000ee`6d1fc788 000000ee`6d1fc7e0 000000ee`6d1fc7d0 :
nt!IopCreateFile+0x3d8
0c fffff803`cd752fa3 : ffffc000`1f296870 fffff803`cd9d9fbd ffffd001`c3098be8 00000000`00000000 :
nt!NtCreateFile+0x79
0d 00007fff`69805b74 : 00007fff`487484e6 0000029b`00000003 00000000`0000012e 00000000`00000000 :
nt!KiSystemServiceCopyEnd+0x13
0e 00007fff`487484e6 : 0000029b`00000003 00000000`0000012e 00000000`00000000 00000000`00000000 :
0x00007fff`69805b74
0f 0000029b`00000003 : 00000000`0000012e 00000000`00000000 00000000`00000000 00000000`00000000 :
0x00007fff`487484e6
10 00000000`0000012e : 00000000`00000000 00000000`00000000 00000000`00000000 00000000`00000000 :
0x0000029b`00000003
11 00000000`00000000 : 00000000`00000000 00000000`00000000 00000000`00000000 00000080 00000000`00000000 : 0x12e
```

You can use DML to further explore the code. When you click on the first 00 entry, the **.frame (Set Local Context)** command is used to set the context and then, the **dv (Display Local Variables)** command displays the local variables.

```
0: kd> .frame 0n0;dv /t /v
00 ffffd001`c30981d0 fffff800`7a0fa607 tabletaudiosample!CMiniportWaveRT::NewStream+0x1dc
[c:\data1\threshold\audio\endpointscommon\minwavert.cpp @ 597]
ffffd001`c30982b0 class CMiniportWaveRT * this = 0xffffe001`d1182000
ffffd001`c30982b8 struct IMiniportWaveRTStream ** OutStream = 0xffffe001`d4776d20
ffffd001`c30982c0 struct IPortWaveRTStream * OuterUnknown = 0xffffe001`d4776bc8
ffffd001`c30982c8 unsigned long Pin = 0
ffffd001`c30982d0 unsigned char Capture = 0x00 '
ffffd001`c30982d8 union KSDATAFORMAT * DataFormat = 0xffffe001`cd7609b0
ffffd001`c3098270 struct _GUID signalProcessingMode = {4780004E-7133-41D8-8C74-660DADD2C0EE}
ffffd001`c3098210 long ntStatus = 0n0
ffffd001`c3098218 class CMiniportWaveRTStream * stream = 0x00000000`00000000
```

Section 10: Display processes and threads

In Section 10, you will use debugger commands to display processes and threads.

Process

To change the current process context, use the .process <process> command. The following example

demonstrates how to identify a process and switch context to it.

- Use the `!process` command to display the current process that is involved in playing the sound.

  For more information see **!process**

The output shows that the process is associated with audiodg.exe. If you are still at the breakpoint described in the previous section of this topic, the current process should be associated with the audiodg.exe image.

<- On the host system

```
0: kd> !process
PROCESS ffffe001d147c840
    SessionId: 0  Cid: 10f0    Peb: ee6cf8a000  ParentCid: 0434
    DirBase: d2122000  ObjectTable: ffffc0001f191ac0  HandleCount: <Data Not Accessible>
    Image: audiodg.exe
    VadRoot ffffe001d4222f70 Vads 70 Clone 0 Private 504. Modified 16. Locked 0.
    DeviceMap ffffc00019113080
    Token                             ffffc0001f1d4060
    ElapsedTime                       <Invalid>
    UserTime                          00:00:00.000
    KernelTime                        00:00:00.000
    QuotaPoolUsage[PagedPool]         81632
    QuotaPoolUsage[NonPagedPool]      9704
    Working Set Sizes (now,min,max)  (2154, 1814, 2109) (8616KB, 7256KB, 8436KB)
    PeakWorkingSetSize                2101
    VirtualSize                       2097192 Mb
    PeakVirtualSize                   2097192 Mb
    PageFaultCount                    2336
    MemoryPriority                    BACKGROUND
    BasePriority                      8
    CommitCharge                      1573

        THREAD ffffe001d173e840  Cid 10f0.1dac  Teb: 000000ee6cf8b000 Win32Thread: ffffe001d1118cf0 WAIT:
(UserRequest) UserMode Non-Alertable
            ffffe001d16c4dd0  NotificationEvent
            ffffe001d08b0840  ProcessObject

        THREAD ffffe001ceb77080  Cid 10f0.16dc  Teb: 000000ee6cf8d000 Win32Thread: 0000000000000000 WAIT:
(WrQueue) UserMode Alertable
            ffffe001cf2d1840  QueueObject

        THREAD ffffe001d112c840  Cid 10f0.0a4c  Teb: 000000ee6cf8f000 Win32Thread: 0000000000000000 WAIT:
(WrQueue) UserMode Alertable
            ffffe001cf2d1840  QueueObject

        THREAD ffffe001d16c7840  Cid 10f0.13c4  Teb: 000000ee6cf91000 Win32Thread: 0000000000000000 WAIT:
(WrQueue) UserMode Alertable
            ffffe001cf2d1840  QueueObject

        THREAD ffffe001cec67840  Cid 10f0.0dbc  Teb: 000000ee6cf93000 Win32Thread: 0000000000000000 WAIT:
(WrQueue) UserMode Alertable
            ffffe001d173e5c0  QueueObject

        THREAD ffffe001d1117840  Cid 10f0.1d6c  Teb: 000000ee6cf95000 Win32Thread: 0000000000000000 WAIT:
(WrQueue) UserMode Alertable
            ffffe001d173e5c0  QueueObject

        THREAD ffffe001cdeae840  Cid 10f0.0298  Teb: 000000ee6cf97000 Win32Thread: 0000000000000000 RUNNING on
processor 2
```

Note that one of the threads associated with this process is in the RUNNING state. This thread was supporting the playing of the media clip when the breakpoint was hit.

Use the **!process 0 0** command to display summary information for all processes. In the command output use

CTRL+F to locate the process ID for the process associated with the audiodg.exe image. In the example shown below, the process ID is *ffffe001d147c840*.

Record the process ID associated with audiodg.exe on your PC to use later in this lab. _____

```
...

PROCESS ffffe001d147c840
    SessionId: 0  Cid: 10f0    Peb: ee6cf8a000  ParentCid: 0434
    DirBase: d2122000  ObjectTable: ffffc0001f191ac0  HandleCount: <Data Not Accessible>
    Image: audiodg.exe
...
```

Enter g into the debugger to run the code forward until the media clip is done playing. Then break in to the debugger, by pressing Ctrl+ScrLk (Ctrl+Break) Use the !process command to confirm that you are now running a different process.

```
!process
PROCESS ffffe001cd0ad040
    SessionId: none  Cid: 0004    Peb: 00000000  ParentCid: 0000
    DirBase: 001aa000  ObjectTable: ffffc00017214000  HandleCount: <Data Not Accessible>
    Image: System
    VadRoot ffffe001d402b820 Vads 438 Clone 0 Private 13417. Modified 87866. Locked 64.
    DeviceMap ffffc0001721a070
    Token                             ffffc00017216a60
    ElapsedTime                       05:04:54.716
    UserTime                          00:00:00.000
    KernelTime                        00:00:20.531
    QuotaPoolUsage[PagedPool]         0
    QuotaPoolUsage[NonPagedPool]      0
    Working Set Sizes (now,min,max)  (1720, 50, 450) (6880KB, 200KB, 1800KB)
    PeakWorkingSetSize                15853
    VirtualSize                       58 Mb
    PeakVirtualSize                   74 Mb
    PageFaultCount                    46128
   MemoryPriority                    BACKGROUND
    BasePriority                      8
    CommitCharge                      66

        THREAD ffffe001cd0295c0  Cid 0004.000c  Teb: 0000000000000000 Win32Thread: 0000000000000000 WAIT:
(Executive) KernelMode Non-Alertable
            fffff803cd8e0120  SynchronizationEvent

        THREAD ffffe001cd02a6c0  Cid 0004.0010  Teb: 0000000000000000 Win32Thread: 0000000000000000 WAIT:
(Executive) KernelMode Non-Alertable
            fffff803cd8e0ba0  Semaphore Limit 0x7fffffff
...
```

The output above shows that a different system process of *ffffe001cd0ad040* is running. The image name shows System, not audiodg.exe.

Now use the !process command to switch to the process that was associated with audiodg.exe. In the example, the process ID is *ffffe001d147c840*. Substitute the process ID in the example with your process ID, that you recorded earlier.

```
0: kd> !process  ffffe001d147c840
PROCESS ffffe001d147c840
    SessionId: 0  Cid: 10f0    Peb: ee6cf8a000  ParentCid: 0434
    DirBase: d2122000  ObjectTable: ffffc0001f191ac0  HandleCount: <Data Not Accessible>
    Image: audiodg.exe
    VadRoot ffffe001d4222f70 Vads 60 Clone 0 Private 299. Modified 152. Locked 0.
    DeviceMap ffffc00019113080
```

```
    Token                             ffffc0001f1d4060
    ElapsedTime                       1 Day 01:53:14.490
    UserTime                          00:00:00.031
    KernelTime                        00:00:00.031
    QuotaPoolUsage[PagedPool]         81552
    QuotaPoolUsage[NonPagedPool]      8344
    Working Set Sizes (now,min,max)  (1915, 1814, 2109) (7660KB, 7256KB, 8436KB)
    PeakWorkingSetSize                2116
    VirtualSize                       2097189 Mb
    PeakVirtualSize                   2097192 Mb
    PageFaultCount                    2464
    MemoryPriority                    BACKGROUND
    BasePriority                      8
    CommitCharge                      1418

        THREAD ffffe001d173e840  Cid 10f0.1dac  Teb: 000000ee6cf8b000 Win32Thread: ffffe001d1118cf0 WAIT:
(UserRequest) UserMode Non-Alertable
            ffffe001d16c4dd0  NotificationEvent
            ffffe001d08b0840  ProcessObject
        Not impersonating
        DeviceMap                 ffffc00019113080
        Owning Process            ffffe001d147c840       Image:         audiodg.exe
        Attached Process          N/A            Image:         N/A
        Wait Start TickCount      338852         Ticks: 197682 (0:00:51:28.781)
        Context Switch Count      36             IdealProcessor: 0
        UserTime                  00:00:00.015
        KernelTime                00:00:00.000
        Win32 Start Address 0x00007ff7fb928de0
        Stack Init ffffd001c2ec6dd0 Current ffffd001c2ec60c0
        Base ffffd001c2ec7000 Limit ffffd001c2ec1000 Call 0
        Priority 8 BasePriority 8 UnusualBoost 0 ForegroundBoost 0 IoPriority 2 PagePriority 5
        Kernel stack not resident.

        THREAD ffffe001d115c080  Cid 10f0.15b4  Teb: 000000ee6cf9b000 Win32Thread: 0000000000000000 WAIT:
(WrQueue) UserMode Alertable
            ffffe001d0bf0640  QueueObject
        Not impersonating
        DeviceMap                 ffffc00019113080
        Owning Process            ffffe001d147c840       Image:         audiodg.exe
        Attached Process          N/A            Image:         N/A
        Wait Start TickCount      338852         Ticks: 197682 (0:00:51:28.781)
        Context Switch Count      1              IdealProcessor: 0
        UserTime                  00:00:00.000
        KernelTime                00:00:00.000
        Win32 Start Address 0x00007fff6978b350
        Stack Init ffffd001c3143dd0 Current ffffd001c3143520
        Base ffffd001c3144000 Limit ffffd001c313e000 Call 0
        Priority 8 BasePriority 8 UnusualBoost 0 ForegroundBoost 0 IoPriority 2 PagePriority 5
        Kernel stack not resident.

        THREAD ffffe001d3a27040  Cid 10f0.17f4  Teb: 000000ee6cf9d000 Win32Thread: 0000000000000000 WAIT:
(WrQueue) UserMode Alertable
            ffffe001d173e5c0  QueueObject
        Not impersonating
        DeviceMap                 ffffc00019113080
        Owning Process            ffffe001d147c840       Image:         audiodg.exe
        Attached Process          N/A            Image:         N/A
        Wait Start TickCount      518918         Ticks: 17616 (0:00:04:35.250)
        Context Switch Count      9              IdealProcessor: 1
        UserTime                  00:00:00.000
        KernelTime                00:00:00.000
        Win32 Start Address 0x00007fff6978b350
        Stack Init ffffd001c70c6dd0 Current ffffd001c70c6520
        Base ffffd001c70c7000 Limit ffffd001c70c1000 Call 0
        Priority 9 BasePriority 8 UnusualBoost 0 ForegroundBoost 0 IoPriority 2 PagePriority 5
        Kernel stack not resident.
```

Because this code is not active, all of the threads are in WAIT state, as expected.

Threads

The commands to view and set threads are very similar to those of processes. Use the **!thread** command to view threads. Use **.thread** to set the current threads.

To explore threads associated with the media player, play the media clip again. If the breakpoint described in the previous section is still in place, you will stop in the context of audiodg.exe.

Use the !thread -1 0 to display brief information for the current thread. This shows the thread address, the thread and process IDs, the thread environment block (TEB) address, the address of the Win32 function (if any) the thread was created to run, and the thread's scheduling state.

```
0: kd> !thread -1 0
THREAD ffffe001d3a27040  Cid 10f0.17f4  Teb: 000000ee6cf9d000 Win32Thread: 0000000000000000 RUNNING on
processor 0
```

To view more information about the thread that is running, type **!thread**. Information similar to the following should be displayed.

```
0: kd> !thread
THREAD ffffe001d3a27040  Cid 10f0.17f4  Teb: 000000ee6cf9d000 Win32Thread: 0000000000000000 RUNNING on
processor 0
IRP List:
    ffffe001d429e580: (0006,02c8) Flags: 000008b4  Mdl: 00000000
Not impersonating
DeviceMap                 ffffc00019113080
Owning Process            ffffe001d147c840       Image:         audiodg.exe
Attached Process          N/A            Image:         N/A
Wait Start TickCount      537630         Ticks: 0
Context Switch Count      63             IdealProcessor: 1
UserTime                  00:00:00.000
KernelTime                00:00:00.015
Win32 Start Address 0x00007fff6978b350
Stack Init ffffd001c70c6dd0 Current ffffd001c70c6520
Base ffffd001c70c7000 Limit ffffd001c70c1000 Call 0
Priority 8 BasePriority 8 UnusualBoost 0 ForegroundBoost 0 IoPriority 2 PagePriority 5
Child-SP          RetAddr           : Args to Child
: Call Site
ffffd001`c70c62a8 fffff800`7a0fa607 : ffffe001`d4aec5c0 ffffe001`cdefd3d8 ffffe001`d4aec5c0 ffffe001`cdefd390
: tabletaudiosample!CMiniportWaveRT::NewStream [c:\data1\threshold\audio\endpointscommon\minwavert.cpp @ 562]
ffffd001`c70c62b0 fffff800`7a0fb2c3 : 00000000`00000000 ffffe001`d429e580 ffffe001`d4ea47b0 ffffe001`cdefd3d8
: portcls!CPortPinWaveRT::Init+0x2e7
ffffd001`c70c6340 fffff800`7a0fc7f9 : ffffe001`d4aec430 00000000`00000000 ffffe001`d10b8380 ffffe001`d429e580
: portcls!CPortFilterWaveRT::NewIrpTarget+0x193
ffffd001`c70c63c0 fffff800`7a180552 : 00000000`00000000 ffffe001`d10b8380 ffffe001`d429e580 ffffe001`d4565600
: portcls!xDispatchCreate+0xd9
ffffd001`c70c6450 fffff800`7a109a9a : ffffe001`d10b84d0 ffffe001`d10b8380 00000000`00000000 ffffe001`00000000
: ks!KsDispatchIrp+0x272
ffffd001`c70c6510 fffff800`7bd314b1 : ffffe001`d429e580 ffffd001`c70c6590 ffffe001`d429e800 ffffe001`ce80da70
: portcls!DispatchCreate+0x7a
ffffd001`c70c6540 fffff803`cda1bfa8 : 00000000`00000025 00000000`00000000 00000000`00000000 ffffe001`d429e580
: ksthunk!CKernelFilterDevice::DispatchIrp+0xf9
ffffd001`c70c65a0 fffff803`cda7b306 : 00000000`000002fc ffffe001`d5e0d510 00000000`00000000 ffffe001`d3341bd0
: nt!IopParseDevice+0x7c8
ffffd001`c70c6770 fffff803`cda12916 : 00000000`000002fc ffffd001`c70c68d0 ffffe001`d3341ba0 fffff803`cda7b250
: nt!IopParseFile+0xb6
ffffd001`c70c67d0 fffff803`cda1131c : ffffe001`ceb6c601 ffffd001`c70c69e0 00000000`00000040 ffffe001`cd127dc0
: nt!ObpLookupObjectName+0x776
ffffd001`c70c6970 fffff803`cd9fedb8 : ffff8ab8`00000001 ffffe001`d5e0d510 00000000`00000000 00000000`00000000
: nt!ObOpenObjectByNameEx+0x1ec
ffffd001`c70c6a90 fffff803`cd9fe919 : 000000ee`6d37c6e8 00000004`6d37c500 000000ee`6d37c5f0 000000ee`6d37c5e0
: nt!IopCreateFile+0x3d8
ffffd001`c70c6b40 fffff803`cd752fa3 : fffff6fb`7da05360 fffff6fb`40a6c0a8 fffff681`4d815760 ffff8ab8`92895e23
: nt!NtCreateFile+0x79
ffffd001`c70c6bd0 00007fff`69805b74 : 00007fff`487484e6 0000029b`00000003 00000000`0000012e 00000000`00000000
: nt!KiSystemServiceCopyEnd+0x13 (TrapFrame @ ffffd001`c70c6c40)
000000ee`6d37c568 00007fff`487484e6 : 0000029b`00000003 00000000`0000012e 00000000`00000000 00000000`00000000
: 0x00007fff`69805b74
000000ee`6d37c570 0000029b`00000003 : 00000000`0000012e 00000000`00000000 00000000`00000000 00000000`00000000
: 0x00007fff`487484e6
000000ee`6d37c578 00000000`0000012e : 00000000`00000000 00000000`00000000 00000000`00000000 00000000`00000080
: 0x0000029b`00000003
000000ee`6d37c580 00000000`00000000 : 00000000`00000000 00000000`00000000 00000000`00000080 00000000`00000000
: 0x12e
```

Use the k command to view the call stack associated with the thread.

```
0: kd> k
# Child-SP          RetAddr           Call Site
00 ffffd001`c70c62a8 fffff800`7a0fa607 tabletaudiosample!CMiniportWaveRT::NewStream
[c:\data1\threshold\audio\endpointscommon\minwavert.cpp @ 562]
01 ffffd001`c70c62b0 fffff800`7a0fb2c3 portcls!CPortPinWaveRT::Init+0x2e7
02 ffffd001`c70c6340 fffff800`7a0fc7f9 portcls!CPortFilterWaveRT::NewIrpTarget+0x193
03 ffffd001`c70c63c0 fffff800`7a180552 portcls!xDispatchCreate+0xd9
04 ffffd001`c70c6450 fffff800`7a109a9a ks!KsDispatchIrp+0x272
05 ffffd001`c70c6510 fffff800`7bd314b1 portcls!DispatchCreate+0x7a
06 ffffd001`c70c6540 fffff803`cda1bfa8 ksthunk!CKernelFilterDevice::DispatchIrp+0xf9
07 ffffd001`c70c65a0 fffff803`cda7b306 nt!IopParseDevice+0x7c8
08 ffffd001`c70c6770 fffff803`cda12916 nt!IopParseFile+0xb6
09 ffffd001`c70c67d0 fffff803`cda1131c nt!ObpLookupObjectName+0x776
0a ffffd001`c70c6970 fffff803`cd9fedb8 nt!ObOpenObjectByNameEx+0x1ec
0b ffffd001`c70c6a90 fffff803`cd9fe919 nt!IopCreateFile+0x3d8
0c ffffd001`c70c6b40 fffff803`cd752fa3 nt!NtCreateFile+0x79
0d ffffd001`c70c6bd0 00007fff`69805b74 nt!KiSystemServiceCopyEnd+0x13
0e 000000ee`6d37c568 00007fff`487484e6 0x00007fff`69805b74
0f 000000ee`6d37c570 0000029b`00000003 0x00007fff`487484e6
10 000000ee`6d37c578 00000000`0000012e 0x0000029b`00000003
11 000000ee`6d37c580 00000000`00000000 0x12e
```

Enter g into the debugger to run the code forward until the media clip is done playing. Then break in to the debugger, by pressing Ctrl - ScrLk (Ctrl-Break) Use the !thread command to confirm that you are now running a different thread.

```
0: kd> !thread
THREAD ffffe001ce80b840  Cid 17e4.01ec  Teb: 00000071fa9b9000 Win32Thread: ffffe001d41690d0 RUNNING on
processor 0
Not impersonating
DeviceMap                 ffffc0001974e2c0
Owning Process            ffffe001d1760840       Image:         rundll32.exe
Attached Process          N/A            Image:         N/A
Wait Start TickCount      538040         Ticks: 0
Context Switch Count      3181840        IdealProcessor: 0
UserTime                  00:00:08.250
KernelTime                00:00:10.796
Win32 Start Address 0x00007ff6d2f24270
Stack Init ffffd001cd16afd0 Current ffffd001cd16a730
Base ffffd001cd16b000 Limit ffffd001cd165000 Call 0
Priority 8 BasePriority 8 UnusualBoost 0 ForegroundBoost 0 IoPriority 2 PagePriority 5

Child-SP          RetAddr           : Args to Child
: Call Site
fffff803`cf373d18 fffff800`7a202852 : fffff803`cf373e60 00000000`00000001 ffffe001`cf4ed330 00000000`0000ffff
: nt!DbgBreakPointWithStatus
fffff803`cf373d20 fffff803`cd6742c6 : ffffe001`cf4ed2f0 fffff803`cf373e60 00000000`00000001 00000000`0004e4b8
: kdnic!TXSendCompleteDpc+0x142
fffff803`cf373d60 fffff803`cd74d495 : 00000000`00000000 fffff803`cd923180 fffff803`cde1f4b0 fffff901`40669010
: nt!KiRetireDpcList+0x5f6
fffff803`cf373fb0 fffff803`cd74d2a0 : 00000000`00000090 0000000e`0000006a 00000000`00000092 00000000`00000000
: nt!KxRetireDpcList+0x5 (TrapFrame @ fffff803`cf373e70)
ffffd001`cd16a6c0 fffff803`cd74bd75 : 00000000`00000000 fffff803`cd74a031 00000000`00000000 00000000`00000000
: nt!KiDispatchInterruptContinue
ffffd001`cd16a6f0 fffff803`cd74a031 : 00000000`00000000 00000000`00000000 ffffe001`cff4d2a0 fffff803`cd67738e
: nt!KiDpcInterruptBypass+0x25
ffffd001`cd16a700 fffff960`50cdb5a4 : fffff901`400006d0 00000000`00000001 fffff901`40000d60 ffffd001`cd16a9f0
: nt!KiInterruptDispatchNoLockNoEtw+0xb1 (TrapFrame @ ffffd001`cd16a700)
ffffd001`cd16a890 fffff960`50c66b2f : 00000000`00000000 fffff901`40669010 fffff901`42358580 fffff901`40000d60
: win32kfull!Win32FreePoolImpl+0x34
ffffd001`cd16a8c0 fffff960`50c68cd6 : 00000000`00000000 ffffd001`cd16a9f0 fffff901`400006d0 fffff901`400c0460
: win32kfull!EXLATEOBJ::vAltUnlock+0x1f
ffffd001`cd16a8f0 fffff803`cd752fa3 : 00000000`00000000 00000000`00000000 ffffe001`ce80b840 00000000`00000000
: win32kfull!NtGdiAlphaBlend+0x1d16
ffffd001`cd16add0 00007fff`674c1494 : 00007fff`674b1e97 0000a7c6`daee0559 00000000`00000001 0000020b`741f3c50
: nt!KiSystemServiceCopyEnd+0x13 (TrapFrame @ ffffd001`cd16ae40)
00000071`fa74c9a8 00007fff`674b1e97 : 0000a7c6`daee0559 00000000`00000001 0000020b`741f3c50 00000000`00ffffff
: 0x00007fff`674c1494
00000071`fa74c9b0 0000a7c6`daee0559 : 00000000`00000001 0000020b`741f3c50 00000000`00ffffff 00000000`00000030
: 0x00007fff`674b1e97
00000071`fa74c9b8 00000000`00000001 : 0000020b`741f3c50 00000000`00ffffff 00000000`00000030 00000000`01010bff
: 0x0000a7c6`daee0559
00000071`fa74c9c0 0000020b`741f3c50 : 00000000`00ffffff 00000000`00000030 00000000`01010bff 00000000`00000000
: 0x1
00000071`fa74c9c8 00000000`00ffffff : 00000000`00000030 00000000`01010bff 00000000`00000000 00000000`000000c0
: 0x0000020b`741f3c50
00000071`fa74c9d0 00000000`00000030 : 00000000`01010bff 00000000`00000000 00000000`000000c0 00000000`00000030
: 0xffffff
00000071`fa74c9d8 00000000`01010bff : 00000000`00000000 00000000`000000c0 00000000`00000030 00000071`00000030
: 0x30
00000071`fa74c9e0 00000000`00000000 : 00000000`000000c0 00000000`00000030 00000071`00000030 00000071`01ff8000
: 0x1010bff
```

The image name is rundll32.exe, which is indeed not the image name associated with playing the media clip.

Note To set the current thread, type .thread <thread number>.

For more information about threads and processes, see the following references:

Threads and Processes

Changing Contexts

Section 11: IRQL, registers, and disassembly

View the saved IRQL

In Section 11, you will display the IRQL, and the contents of the regsisters.

<- On the host system

The interrupt request level (IRQL) is used to manage the priority of interrupt servicing. Each processor has an IRQL setting that threads can raise or lower. Interrupts that occur at or below the processor's IRQL setting are masked and will not interfere with the current operation. Interrupts that occur above the processor's IRQL setting take precedence over the current operation. The **!irql** extension displays the interrupt request level (IRQL) on the current processor of the target computer before the debugger break occurred. When the target computer breaks into the debugger, the IRQL changes, but the IRQL that was effective just before the debugger break is saved and is displayed by !irql.

```
0: kd> !irql
Debugger saved IRQL for processor 0x0 -- 2 (DISPATCH_LEVEL)
```

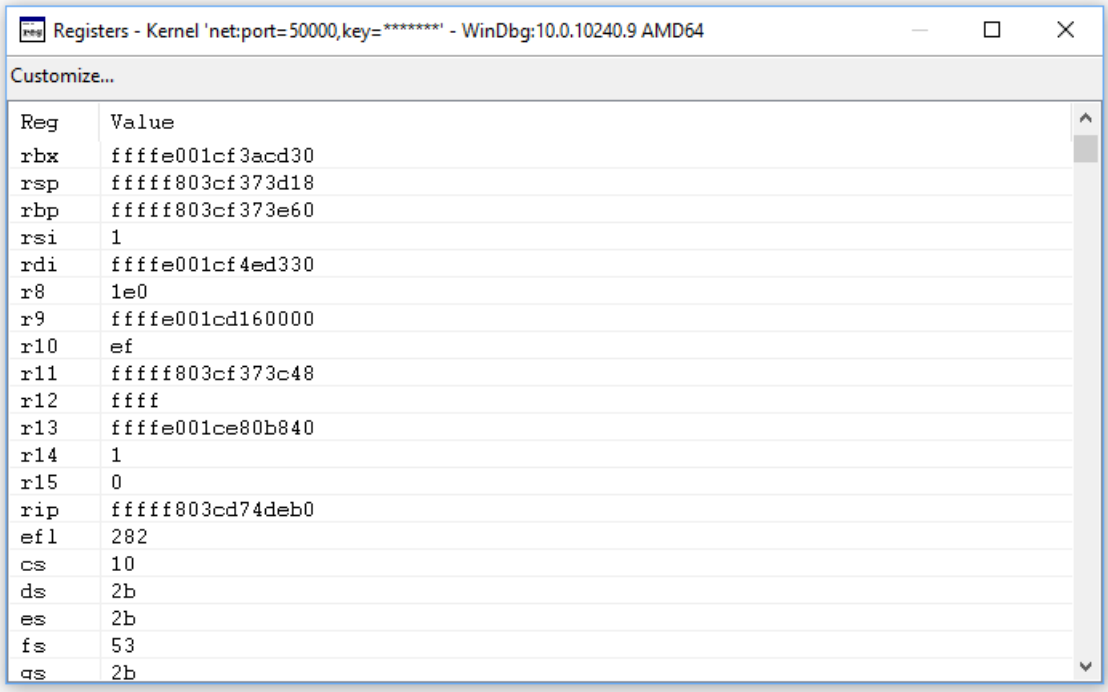
<View the registers and disassembly

View the registers

Display the contents of the registers for the current thread on the current processor by using the **r (Registers)** command.

```
0: kd> r
rax=000000000000c301 rbx=ffffe00173eed880 rcx=0000000000000001
rdx=000000d800000000 rsi=ffffe00173eed8e0 rdi=ffffe00173eed8f0
rip=fffff803bb757020 rsp=ffffd001f01f8988 rbp=ffffe00173f0b620
 r8=000000000000003e  r9=ffffe00167a4a000 r10=000000000000001e
r11=ffffd001f01f88f8 r12=0000000000000000 r13=ffffd001f01efdc0
r14=0000000000000001 r15=0000000000000000
iopl=0         nv up ei pl nz na pe nc
cs=0010  ss=0018  ds=002b  es=002b  fs=0053  gs=002b             efl=00000202
nt!DbgBreakPointWithStatus:
fffff803`bb757020 cc              int     3
```

Alternatively, you can display the contents of the registers by clicking **View** > **Registers**.

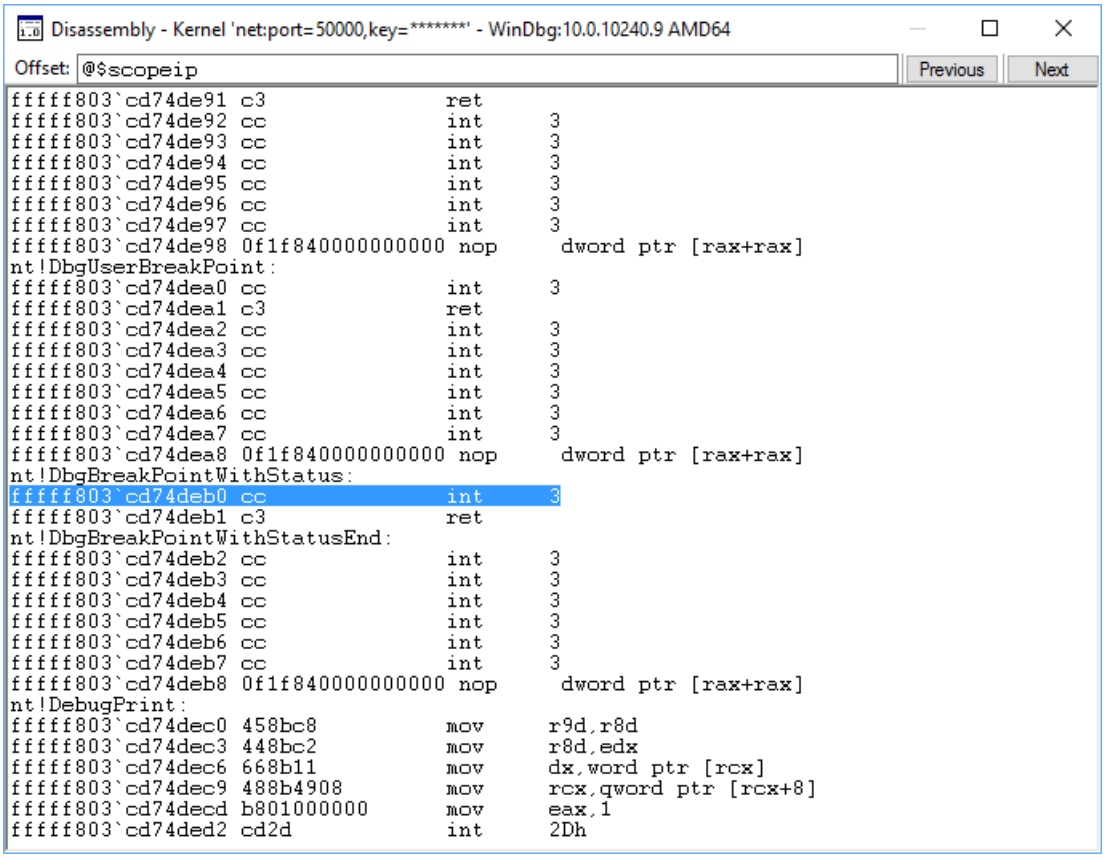


Viewing the contents of the registers can be helpful when stepping through assembly language code execution and in other scenarios. For more information see **r (Registers)**.

For information about contents of the register, see x86 Architecture and x64 Architecture.

Disassembly

You can disassemble the code that is under execution to view the assembly language code that is being run by clicking **View** > **Disassembly**.



For more information about assembly language disassembly, see Annotated x86 Disassembly and Annotated x64 Disassembly.

Section 12: Work with memory

In Section 12, you will use debugger commands to display the contents of memory.

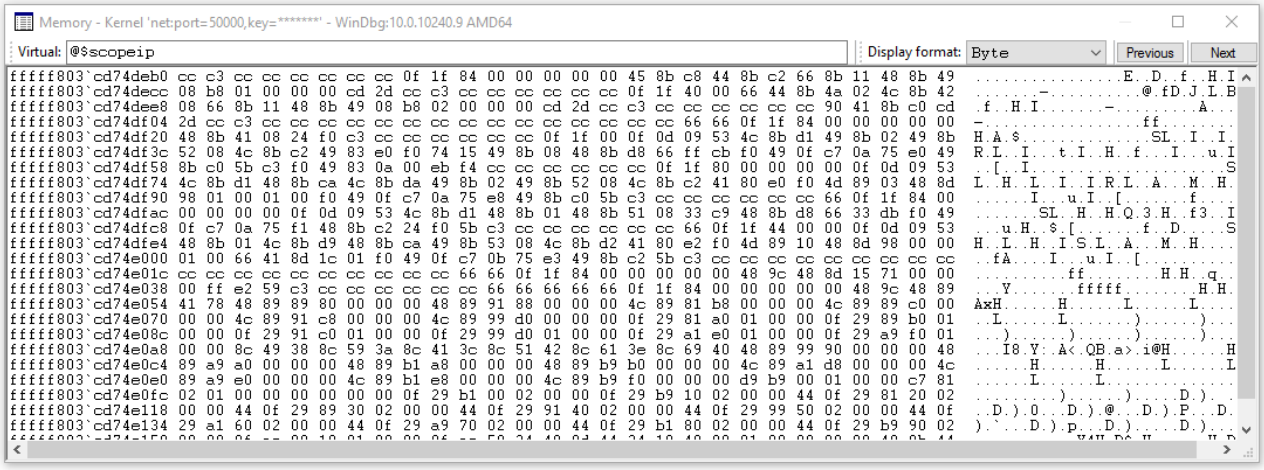
View memory

You may need to examine memory to identify an issue or to inspect variables, pointers, and so on. You can display memory by typing one of the following **d* <address>** commands.

db	Displays data in byte values and ASCII characters.
dd	Displays data as double wide words (4 bytes).
du	Displays data as Unicode characters.
dw	Displays data as word values (2 bytes) and ASCII characters.

Note If you attempt to display an invalid address, its contents are shown as question marks (?).

Alternatively, you can view the memory by clicking **View** > **Memory**. Use the **Display format** pull down to change how the memory is displayed.



1. To view data associated with the volume control, set a breakpoint to fire on the PropertyHandlerAudioEngineVolumeLevel routine using the bm command. Before we set the new breakpoint, we will clear all of the previous breakpoints using bc *.

```
kd> bc *
```

2. Set a breakpoint to fire on the PropertyHandlerAudioEngineVolumeLevel routine using the bm command.

```
kd> bm tabletaudiosample!CMiniportWaveRT::SetDeviceChannelVolume
  1: fffff80f`02c3a4b0 @!"tabletaudiosample!CMiniportWaveRT::SetDeviceChannelVolume"
```

3. List the breakpoints to confirm that the breakpoint is set properly.

```
kd> bl
  1: fffff80f`02c3a4b0 @!"tabletaudiosample!CMiniportWaveRT::SetDeviceChannelVolume"
```

4. Use the **g** command to restart code execution.

   On the target system adjust the volume in the system tray. This will cause the breakpoint to fire.

```
Breakpoint 1 hit
tabletaudiosample!CMiniportWaveRT::SetDeviceChannelVolume:
fffff80f`02c3a4b0 44894c2420      mov     dword ptr [rsp+20h],r9d
```

5. Use the **View**> **Local** menu item to display local variables. Note the current value of the IVolume variable.

6. You can display the data type and the current value for the IVolume variable in the sample code by typing the **dt** command and the name of the variable.

```
kd> dt lVolume
Local var @ 0xa011ea50 Type long
0n-6291456
```

7. The breakpoint is hit on entering SetDeviceChannelVolume.

```
STDMETHODIMP_(NTSTATUS) CMiniportWaveRT::SetDeviceChannelVolume(_In_  ULONG _ulNodeId, _In_ UINT32
_uiChannel, _In_  LONG  _Volume)
{
    NTSTATUS ntStatus = STATUS_INVALID_DEVICE_REQUEST;

    PAGED_CODE ();

    DPF_ENTER(("[CMiniportWaveRT::SetEndpointChannelVolume]"));
    IF_TRUE_ACTION_JUMP(_ulNodeId != KSNODE_WAVE_AUDIO_ENGINE, ntStatus =
STATUS_INVALID_DEVICE_REQUEST, Exit);

    // Snap the volume level to our range of steppings.
    LONG lVolume = VOLUME_NORMALIZE_IN_RANGE(_Volume);

    ntStatus = SetChannelVolume(_uiChannel, lVolume);
Exit:
    return ntStatus;
}
```

8. Attempt to display the value at the memory location of IVolume by using the **dt (Display Type)** command.

```
kd> dt dt lVolume
Local var @ 0xffffb780b7eee664 Type long
0n0
```

   Because the variable is yet to be defined, it does not contain information.

9. Press F10 to run forward to the last line of code in SetDeviceChannelVolume.

```
    return ntStatus;
```

10. Display the value at the memory location of IVolume by using the **dt (Display Type)** command.

```
kd> dt lVolume
Local var @ 0xffffb780b7eee664 Type long
0n-6291456
```

Now that the variable is active, a value of 6291456 is displayed in this example.

11. You can also display the memory location of lVolume by using the **? (Evaluate Expression)** command.

```
kd> ? lVolume
Evaluate expression: -79711507126684 = ffffb780`b7eee664
```

12. The address shown, *ffffb780`b7eee664* is the address of the lVolume variable. Use the dd command to display the contents of memory at that location.

```
kd>  dd ffffb780`b7eee664
ffffb780`b7eee664  ffa00000 00000018 00000000 c52d7008
ffffb780`b7eee674  fffc98e e0495756 fffff80e c52d7008
ffffb780`b7eee684  fffc98e 00000000 fffff80e 00000000
ffffb780`b7eee694  fffc98e ffa00000 ffffb780 b7eee710
ffffb780`b7eee6a4  ffffb780 00000000 00000000 c7477260
ffffb780`b7eee6b4  fffc98e b7eee7a0 ffffb780 b7eee6f0
ffffb780`b7eee6c4  ffffb780 e04959ca fffff80e 00000000
ffffb780`b7eee6d4  00000000 00000028 00000000 00000002
```

13. You can display the first four bytes of an address by specifying the range parameter L4.

```
kd> dd ffffb780`b7eee664 l4
ffffb780`b7eee664  ffa00000 00000018 00000000 c52d7008
```

14. To see the different types of memory output displayed, type the **du**, **da** and **db** commands.

```
kd> du ffffb780`b7eee664
ffffb780`b7eee664  ""

kd> a ffffb780`b7eee664
ffffb780`b7eee664  ""

kd> db 0xffffae015ff97664
ffffae01`5ff97664  00 80 bc ff 18 00 00 00-00 00 00 00 08 50 e0 51  .............P.Q
ffffae01`5ff97674  00 c0 ff ff 56 57 da 56-0e f8 ff ff 08 50 e0 51  ....VW.V.....P.Q
ffffae01`5ff97684  00 c0 ff ff 00 00 00 00-0e f8 ff ff 00 00 00 00  ................
ffffae01`5ff97694  00 c0 ff ff aa 80 bc ff-01 ae ff ff 10 77 f9 5f  .............w._
ffffae01`5ff976a4  01 ae ff ff 40 00 00 00-00 e6 ff ff 10 dc 30 55  ....@.........0U
ffffae01`5ff976b4  00 c0 ff ff a0 77 f9 5f-01 ae ff ff f0 76 f9 5f  .....w._.....v._
ffffae01`5ff976c4  01 ae ff ff ca 59 da 56-0e f8 ff ff 00 00 00 00  .....Y.V........
ffffae01`5ff976d4  00 00 00 00 28 00 00 00-00 00 00 00 02 00 00 00  ....(...........
```

Use the df float option to display data as single-precision floating-point numbers (4 bytes).

```
df ffffb780`b7eee664
ffffb780`b7eee664          -1.#QNAN   3.3631163e-044               0        -2775.002
ffffb780`b7eee674          -1.#QNAN  -5.8032637e+019         -1.#QNAN        -2775.002
ffffb780`b7eee684          -1.#QNAN                0         -1.#QNAN                0
ffffb780`b7eee694          -1.#QNAN         -1.#QNAN         -1.#QNAN  -2.8479408e-005
```

Write to memory

Similar to the commands that are used for reading memory, you can use the e* commands to change memory contents.

COMMAND	DESCRIPTION
ea	ASCII string (not NULL-terminated)
eu	Unicode string (not NULL-terminated
ew	Word values (2 bytes)
eza	NULL-terminated ASCII string
ezu	NULL-terminated Unicode string
eb	Byte values
ed	Double-word values (4 bytes)

The following example shows how to overwrite memory.

1. First, locate the address of the lVolume that is used in the sample code.

```
kd> ? lVolume
Evaluate expression: -79711507126684 = ffffb780`b7eee664
```

2. Overwrite that memory address with new characters using the **eb** command.

```
kd> eb 0xffffb780`b7eee664 11 11 11 11 11
```

3. Display the memory location to confirm that the characters have been overwritten by typing the **db** command.

```
kd> db 0xffffb780`b7eee664
ffffb780`b7eee664  11 11 11 11 11 00 00 00-00 00 00 00 08 70 2d c5  .............p-.
ffffb780`b7eee674  8e c9 ff ff 56 57 49 e0-0e f8 ff ff 08 70 2d c5  ....VWI......p-.
ffffb780`b7eee684  8e c9 ff ff 00 00 00 00-0e f8 ff ff 00 00 00 00  ................
ffffb780`b7eee694  8e c9 ff ff 00 00 a0 ff-80 b7 ff ff 10 e7 ee b7  ................
ffffb780`b7eee6a4  80 b7 ff ff 00 00 00 00-00 00 00 00 60 72 47 c7  ............`rG.
ffffb780`b7eee6b4  8e c9 ff ff a0 e7 ee b7-80 b7 ff ff f0 e6 ee b7  ................
ffffb780`b7eee6c4  80 b7 ff ff ca 59 49 e0-0e f8 ff ff 00 00 00 00  .....YI.........
ffffb780`b7eee6d4  00 00 00 00 28 00 00 00-00 00 00 00 02 00 00 00  ....(...........
```

Alternatively, you can modify the contents of the memory in a watch or locals window. For the watch window, you may see variables that are out of context of the current frame. Modifying them is not relevant if they are not in context.

Section 13: End the WinDbg session

<-On the host system

To end a user-mode debugging session, return the debugger to dormant mode, and set the target application to run again, enter the **qd** (Quit and Detach) command.

Be sure and use the **g** command to let the target computer run code, so that it can be used. It also a good idea to clear any break points using **bc ***, so that the target computer won't break and try to connect to the host computer debugger.

```
0: kd> qd
```

For more information, see Ending a Debugging Session in WinDbg in the debugging reference documentation.

Section 14: Windows debugging resources

Additional information is available on Windows debugging. Note that some of these books will use older versions of Windows such as Windows Vista in their examples, but the concepts discussed are applicable to most versions of Windows.

Books

- *Advanced Windows Debugging* by Mario Hewardt and Daniel Pravat
- *Inside Windows Debugging: A Practical Guide to Debugging and Tracing Strategies in Windows®* by Tarik Soulami
- *Windows Internals* by Mark E. Russinovich, David A. Solomon and Alex Ionescu

Video

The Defrag Tools Show WinDbg Episodes 13-29 https://channel9.msdn.com/Shows/Defrag-Tools

Training Vendors:

OSR - https://www.osr.com/

See Also

Getting Started with Windows Debugging

Debugging Using WinDbg Preview

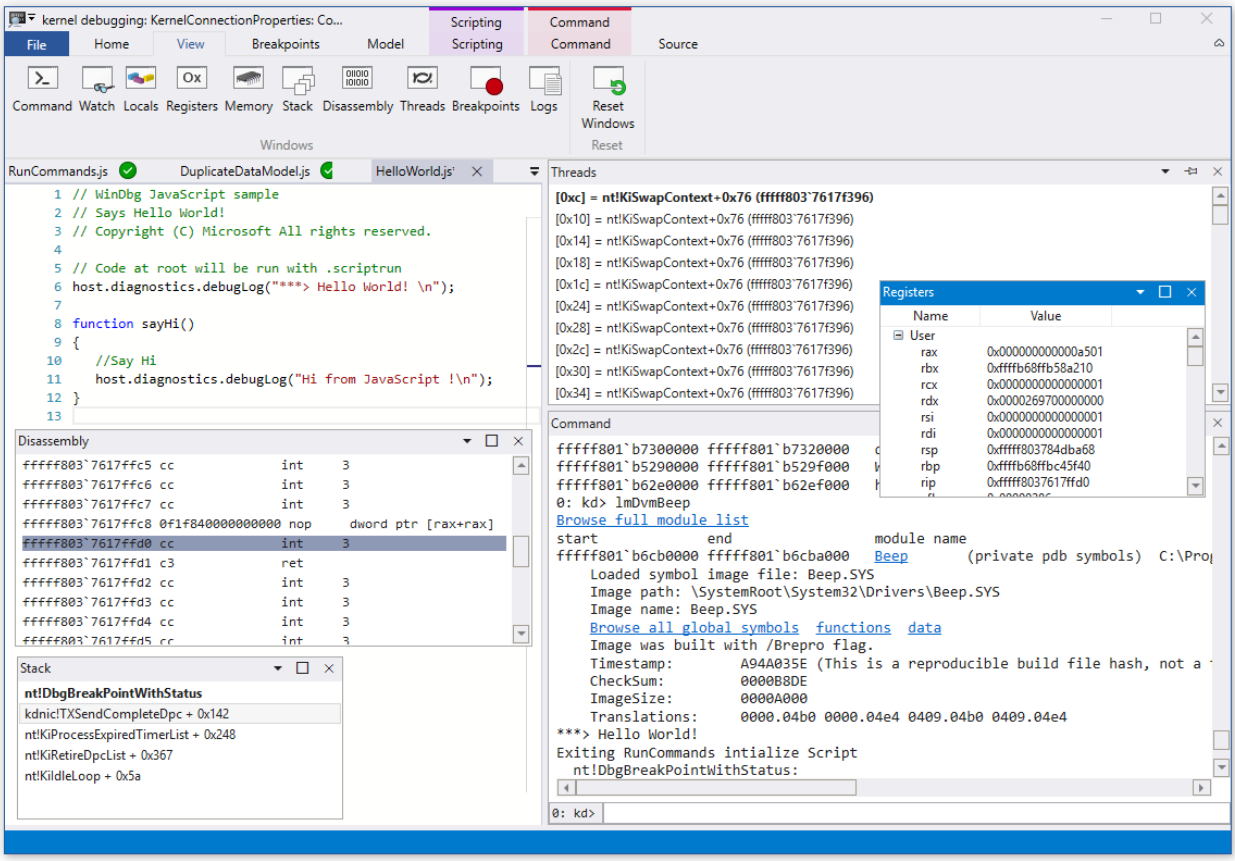
10/25/2019 • 4 minutes to read • Edit Online

WinDbg Preview is a brand-new version of WinDbg with more modern visuals, faster windows, a full-fledged scripting experience, built with the extensible debugger data model front and center. WinDbg Preview is using the same underlying engine as WinDbg today, so all the commands, extensions, and workflows you're used to will still work as they did before.

For the latest news, tips, and tricks from the debugger dev team, refer to the debugger tools team blog.
https://blogs.msdn.microsoft.com/windbg/

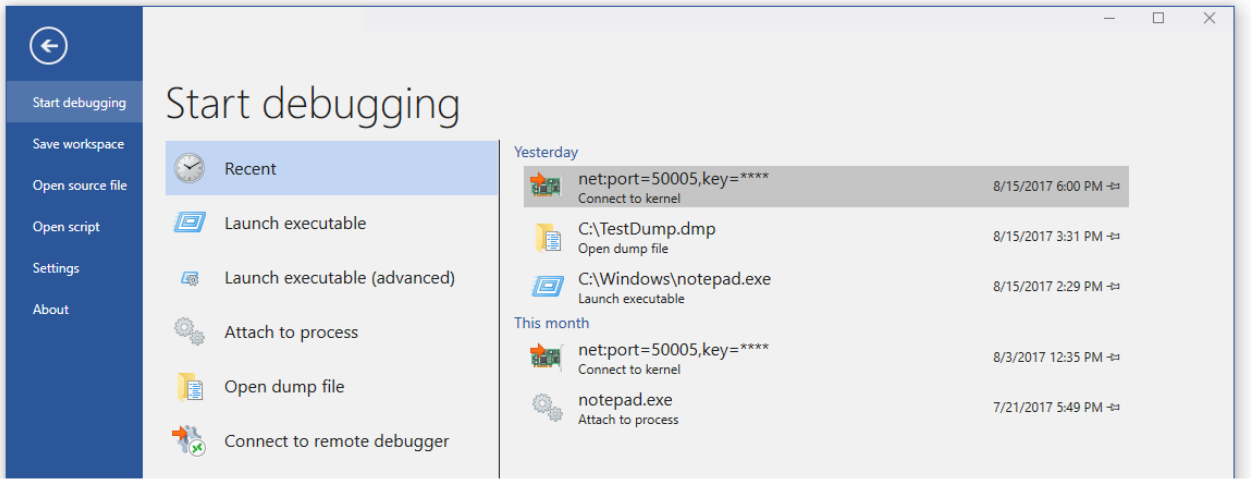
Major Features of WinDbg Preview

Here's some of the most notable things that have changed or are new.



General features

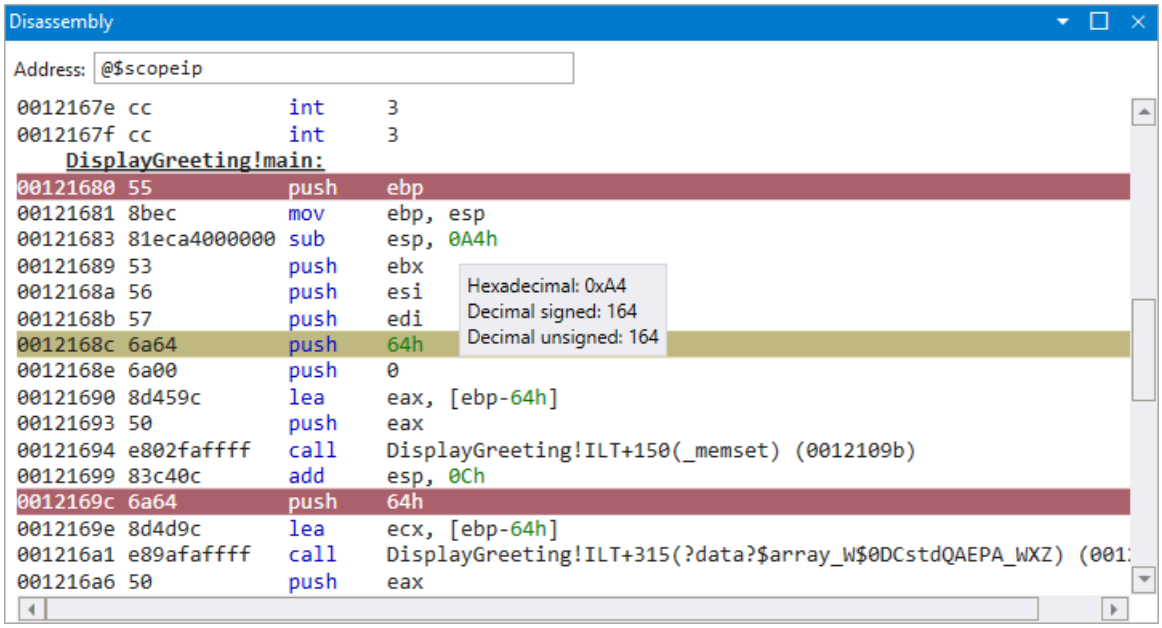
- **Easier Connection Setup and Recall** - The WinDbg Preview includes the ability to recall previous session configuration information.



- **Easy feedback channel** - Your feedback will guide the development effort going forward. For more information, see Providing Feedback
- **Dump file processor detection** -Auto-detects processor architecture for easier managed debugging.
- **Performance Improvements** Windows now load asynchronously and can be canceled - When you run another command, WinDbg Preview will stop the loading of your locals, watch, or other windows.

Windowing improvements

- **Disassembly Window Improvements** - The disassembly window is also improved, the highlight of the current instruction remains where it is when you scroll.



- **Memory window improvements** - The memory window has highlighting and improved scrolling.
- **Locals and watch data model visualization** - The locals and watch windows are both based off of the data model that is used by the dx command. This means the locals and watch windows will benefit from any NatVis or JavaScript extensions you have loaded, and can even support full LINQ queries, just like the dx command.
- **Logs** - This is a under the covers log of the WinDbg Preview internals. It can be viewed for troubleshooting or to monitor long running processes.

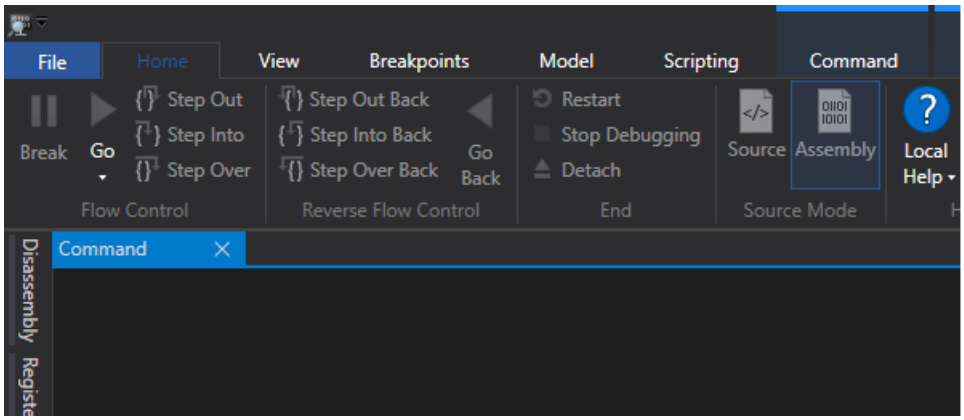
For more information, see WinDbg Preview - View menu.

- **Command window** - Use the command window provides easy access to toggle DML and clear the debugger command window. All current debugger commands are compatible with and continue to work in

WinDbg Preview.

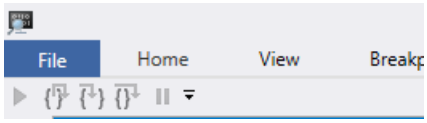
Dark theme

Use **File** > **Settings** to enable the dark theme.



Ribbon Quick Access

Just pin the buttons you use the most and you can collapse the ribbon to save screen real estate.

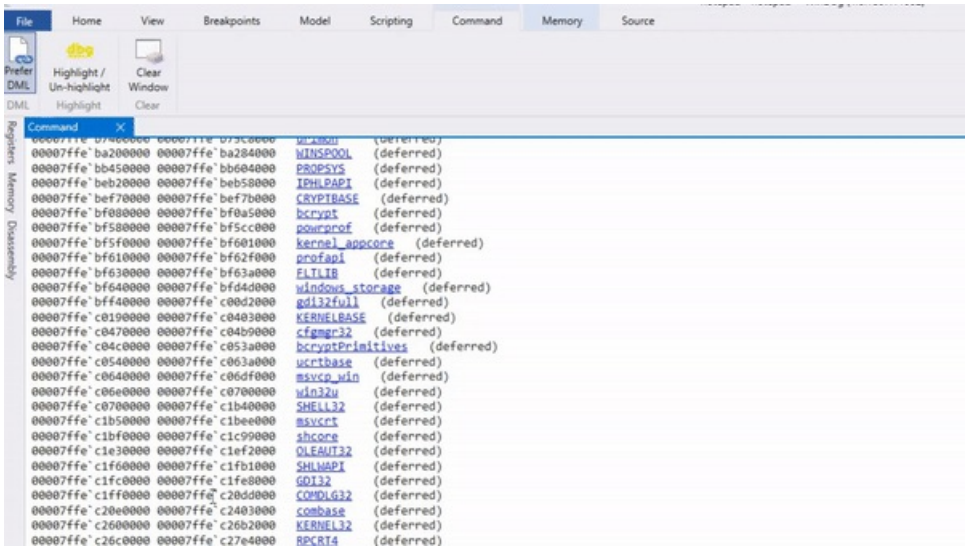


Source Window

The source window has been updated to be much more in line with modern editors.

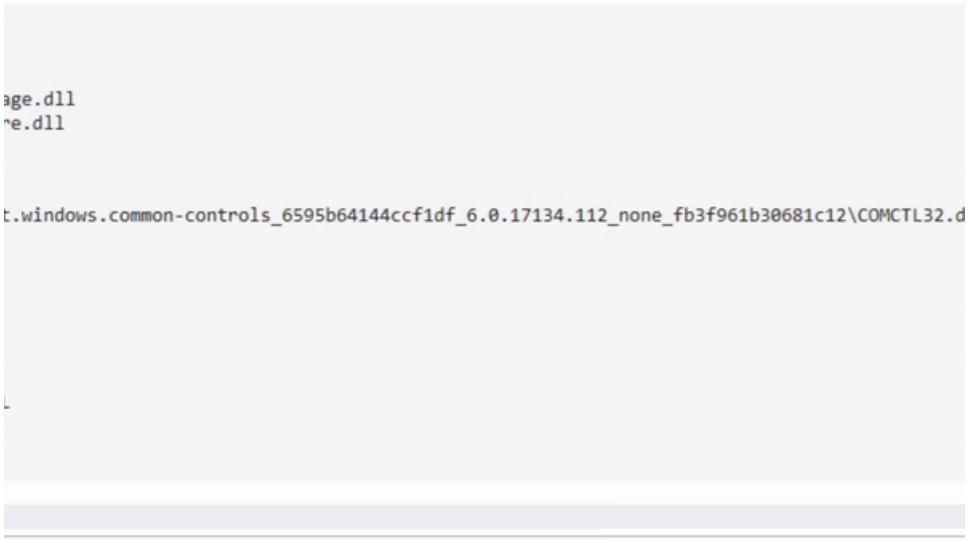
Highlighting

The command window has two new highlighting features. Selecting any text will give a subtle highlight to any other instances of that text. You can then hit "Highlight/Un-highlight" or Ctrl+Alt+H to persist the highlighting.



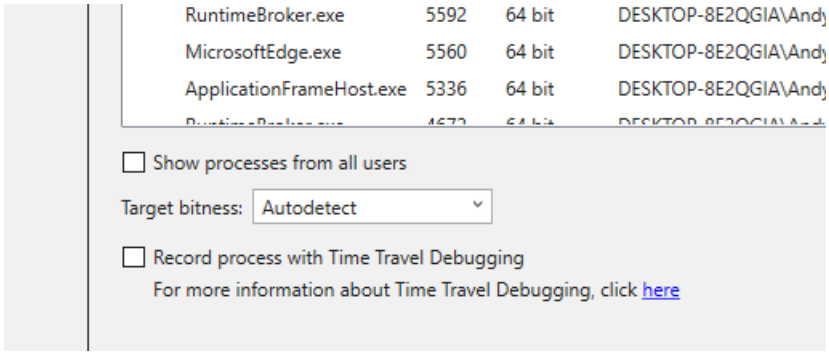
Better keyboard navigation

Just hit Ctrl+Tab and you can easily navigate between windows with just your keyboard.



Integrated Time Travel Debugging (TTD)

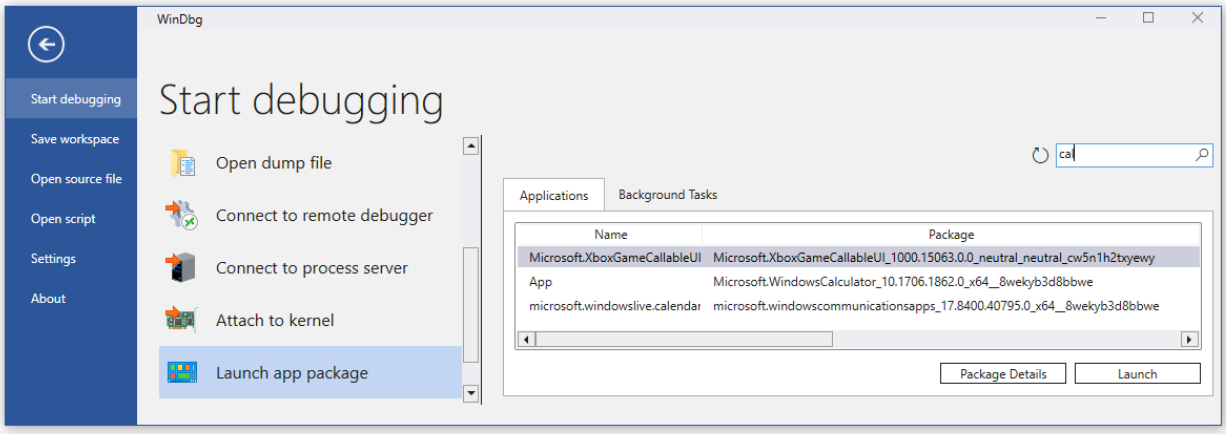
If you need a TTD trace of your application, just check the "Record process with Time Travel Debugging" box when launching or attaching. WinDbgNext will set it up for TTD and open the trace when you're done recording.



For more information, see Time Travel Debugging - Overview.

Debugging App packages

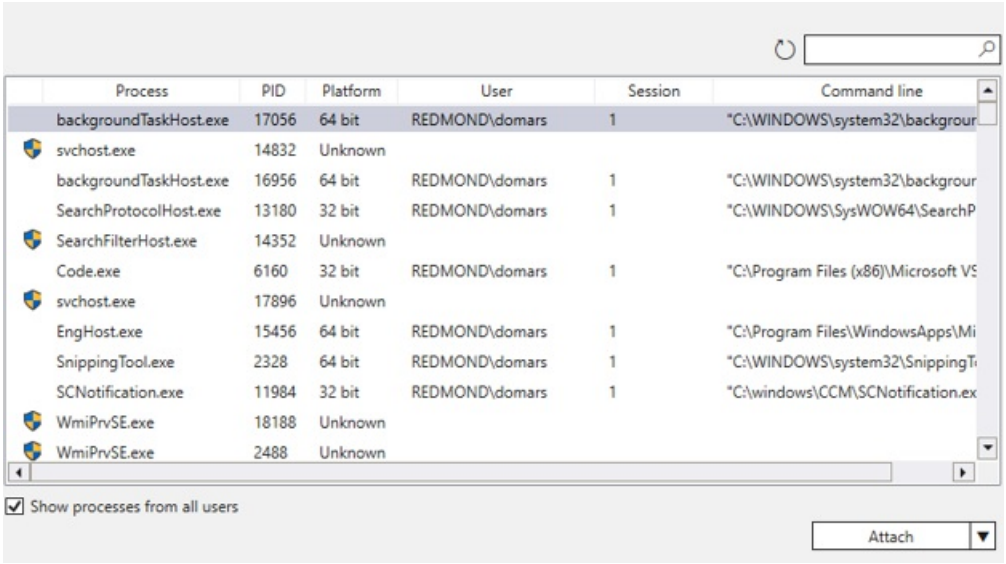
Debugging your universal app or background task is now a single click.



For more information, see Launch App Package.

Attach to a process

The attach dialog is faster, more detailed, and is easier to use.



Enhanced breakpoint tracking

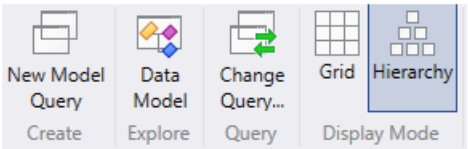
- **Enable/Disable breakpoints** - The breakpoints window shows all your current breakpoints and provides easy access to enabling and disabling them.
- **Hit count** - The breakpoint window keeps a running total of each time the breakpoint is hit.

For more information, see Breakpoints.

Enhanced data model support

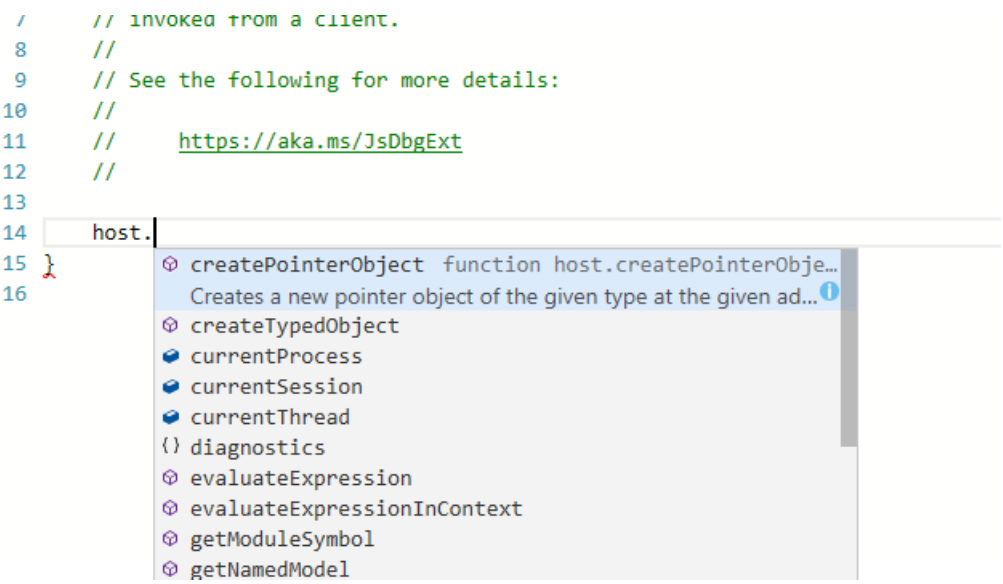
- **Built in data model support** - WinDbg Preview is written with built in data model support and the data model is available through out the debugger.
- **Model window** - The model window gives you an expandable and browsable version of 'dx' and 'dx -g', letting you create powerful tables on-top of your NatVis, JavaScript, and LINQ queries.

For more information, see WinDbg Preview - Data model.



New scripting development UI

- **Script development UI** - There is now a purpose built scripting window to make developing JavaScript and NatVis scripts easier, with error highlighting and IntelliSense.



For more information, see WinDbg Preview - Scripting.

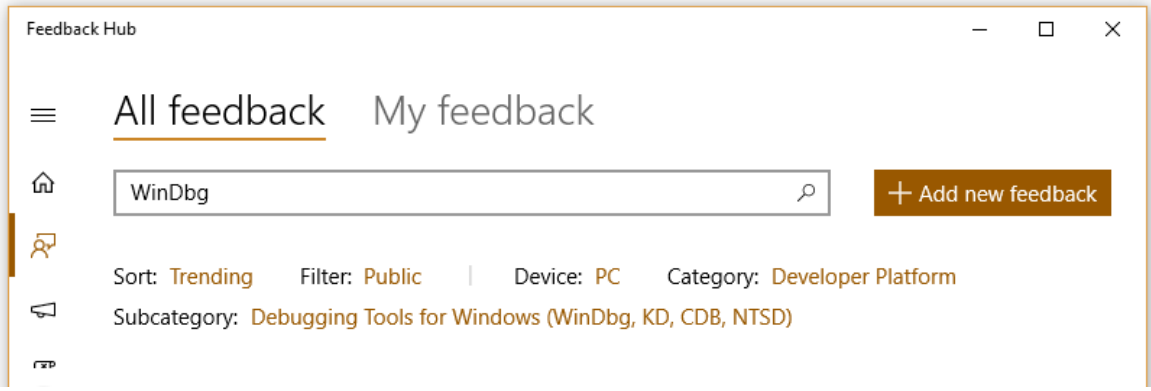
Backwards compatibility

Because the underling debugger engine is the same, all of the previous debugger commands and debugger extensions continue to work.

Providing feedback

Your feedback will help guide WinDbg's development going forward.

- If you have feedback such as a feature that you really want to see or a bug that makes something difficult, use the Feedback Hub.



Videos

Watch these episodes of the Defrag Tools show to see Windbg Preview in action.

- Defrag Tools #182 - Tim, Chad, and Andy go over the basics of WinDbg Preview and some of the features.
- Defrag Tools #183 - Nick, Tim, and Chad use WinDbg Preview and go over a quick demo.
- Defrag Tools #184 - Bill and Andrew walk through the scripting features in WinDbg Preview.
- Defrag Tools #185 - James and Ivette provide and introduction to Time Travel Debugging.
- Defrag Tools #186 - James and JCAB covers advanced Time Travel Debugging.

Next Steps

For information on what's new in the most recent release, see WinDbg Preview - What's New.

Review these topics to install and configure WinDbg Preview.

- WinDbg Preview – Installation
- WinDbg Preview – Command line startup options
- WinDbg Preview – Settings and workspaces
- WinDbg Preview – Keyboard shortcuts

These topics describe how to get connected to the environment that you want to debug.

- WinDbg Preview – Start a user-mode session
- WinDbg Preview – Start a kernel mode session

These topics describe some common tasks, organized by the menu tabs.

- WinDbg Preview – File menu
- WinDbg Preview – Home menu
- WinDbg Preview – View menu
- WinDbg Preview – Breakpoints
- WinDbg Preview – Data model
- WinDbg Preview – Scripting

WinDbg Preview - What's New

9/3/2019 • 16 minutes to read • Edit Online

This topic provides information on what's new in the WinDbg Preview debugger.

1.0.1908.30002

Improvements to TTD Calls objects - Calls queries now include parameter names, types, and values. When querying across traces for function calls you can get fully typed parameters and their values making it easy to filter down results by parameters.

Support for Open Enclave - WinDbg Preview can now debug Open Enclave (OE) applications, you can find directions for how to do this in the Open Enclave documentation.

VS Code Extension - To make it easier to develop for Open Enclave, we've released a basic VS Code extension to enable a quicker inner loop. Variables, Watch, and Call Stack windows all work as well as breakpoints and source windows, any deeper debugging will need to use the console window.

You can find the extension in the VS Code Marketplace and report any issues to our WinDbg Feedback GitHub. Note that while the extension may work for other scenarios, we're only intending on fixing issues related to OE scenarios at this point.

ELF Core Dumps - As part of supporting Open Enclave, WinDbg can open ELF core dumps and binaries as well as DWARF symbols (DWARF 5 is not currently supported) from both Enclaves and Linux applications. When opening a core dump from a non-Windows application, basic windows and commands should all work properly, but most extensions and Windows-specific commands will not work. ELF and DWARF files will be downloaded from symbol servers following the key conventions defined here. Enclaves are the only supported scenario, but we're open to feedback on opening other Linux core dumps.

TTD File format change - We've made a major update to the file format for TTD traces that breaks forward compatibility. Previous versions of WinDbg Preview will not be able to open traces recorded with this (and future) versions of WinDbg Preview, but this (and future) versions will be able to open both new and old traces.

Other changes

- TTD will now use the 64-bit engine for indexing and the appropriate debugger engine bitness for replays to minimize potential memory issues when indexing and SOS issues when replaying.
- Running 'dx' without any parameters will now show the root namespace for easier browsability.
- You can now modify the default symbol and source cache location via the settings menu.
- Improved support for recording AVX-512 (recording of AVX-512 will cause a larger than normal slow-down)
- We've enabled offline licensing

1.0.1905.12001

Improvements to SymSetDiaSession error mitigation - Our fix last month to mitigate the error caused by applications injecting DbgHelp into our process was still not working in some scenarios. We've made improvements to it and will continue to monitor feedback on this error.

Accent color customization - A lot of scenarios need several instances of WinDbg open, and moving back and forth between them can be confusing and take some time to figure out which one is the "right" one. We've added the ability to change the blue accent color to help visually distinguish sessions and make swapping between them easier.

Just click the **View** ribbon and select an option for **Accent color** in the last section. When future sessions are launched from recent targets, the accent color will be persisted as part of the target's workspace.

Source tokenization improvements - The source window now has basic support for tokenizing Rust source files and C++ SEH __try/__except/__finally/__leave.

Coroutine improvements - Improved support for coroutine local variables and certain optimized variables.

Default symbol and source cache setting - Added an option to the settings menu under **Debugging settings** to change the cache location for symbols. **Note** - There's a known issue that making this blank will cause source loading to fail. We'll be adding validation to prevent this from happening in a future release.

-pv fixes - Fixed a bug that may have prevented -pv (non-invasive attach) from working in some circumstances.

1.0.1904.18001

Fix for SymSetDiaSession error - We've had reports for a while of an error that prevents WinDbg Preview from being launched in some situations. There are a few external applications that attempt to inject a version of DbgHelp into our process before we load it. Some of them are using a version of DbgHelp with missing functionality, which causes this error when we attempt to use those features. We've added a fix for this and will be tracking if there are still scenarios in which it occurs.

Font control - We've added settings for controlling font and font size. There are two different settings, one for text windows (mono-spaced windows like disassembly, source, command, etc.) and one for tool windows (locals, stack, etc.). There's still a few areas that aren't affected by these options that we'll be updating in the future.

Highlighting improvements - Persistent highlighting of text in the command window will now also highlight text in the source and notes windows.

Source loading improvements - We've changed how loading source files works. Previously when opening a source file, engine operations like running additional commands weren't possible or were unpredictable. We've changed where the loading occurs to enable better parallelism and more reliable cancellation of source opening operations.

Other changes and bug fixes:

- Added "Go to disassembly" to the context menu of the source window.
- Added a checkbox to "Follow current instruction" in disassembly window.
- Fixed a bug that caused the command window to perform slowly when outputting lots of text.
- Changed page up and page down keys to perform similar to Visual Studio.
- When an ASM file is opened in the source window it will now have basic comment, string, and directive highlighting

1.0.1812.12001

This version includes these updates.

Debugger data model C++ header - There is a new C++ header, DbgModel.h, included as part of the Windows SDK for extending the debugger data model via C++. You can find more information in Debugger Data Model C++ Overview. This release includes a new extension that adds some more "API style" features to the debugger data model that can be accessed through the 'dx' command, JavaScript, and the new DbgModel.h header. This extension extensions the data model to include knowledge about assembly and code execution through the Debugger.Utility.Code namespace, and the local file system through the Debugger.Utility.FileSystem namespace.

Synthetic types extension With this new API extension, we have a new sample up on our GitHub repo here - https://github.com/Microsoft/WinDbg-Samples/tree/master/SyntheticTypes. This JavaScript extension reads basic C header files and defines synthetic type information for the structures and unions defined in the header. Through

the dx command, memory can then be viewed structured as if you had a PDB with type information for those types.

Other changes and bug fixes:

- WinDbg Preview will now more intelligently handle bringing source windows or the disassembly window to the foreground when stepping.
- Re-arranged WinDbgNext's window title to have more important information at the start when kernel debugging.
- The alternating background contrast in the command window should be slightly more noticeable.

1.0.1810.2001

This version includes these updates.

- New Settings dialog that is accessed from the File menu or the Home ribbon.
- Events and exceptions settings dialog. This menu changes how the debugger handles events and exceptions, the equivalent of the 'sx' commands or WinDbg's event filters dialog. Select **Settings** on the home ribbon, then hit "Events and Exceptions" on the left to manage those.
- Improved TTD indexer with better performance. This increases the performance of indexing TTD trace files, making the indexing process much faster (between 2x-10x) while making index files much smaller (~50% smaller). The perf improvements are most noticeable for traces over 4GB in size, or when using a machine with many CPU cores (8+). The new indexer makes it more feasible to debug very large traces (50GB+).
- New *debugArch* launch flag for specifying architecture. WinDbg Preview attempts to launch the debugger engine with the correct bitness to the target to better support debugging managed code. There are circumstances where it can't determine the right bitness or you may want to override what it decides. Use -debugArch x86|amd64 to control the architecture of debugger engine.

Other changes and bug fixes:

- Fixed a bug that would cause black bars to appear on a full screen debugger with a floating window open.
- Fixed a bug that would cause symbol options to be cleared unintentionally.
- Command history is now preserved when launching from recent targets.
- In the data model window, you can now edit values.
- Un-indexed TTD traces will now be more clear that they're un-indexed.
- Improved performance of the locals window
- Added a ribbon button to save the command window logs to a file.
- Added . SelectMany() to the default set of LINQ methods.

1.0.1807.11002

This version includes these updates.

Automatic saving and loading of breakpoints. This is a first step to replace workspaces. We're starting down that route by enabling the saving and loading of breakpoints. Launching something you've debugged previously from the "Recents" tab in the file menu will now load the breakpoints from that session. The plan is to expand this functionality to preserve more information across sessions. Hardware breakpoints (ba) and other various properties on breakpoints like thread and process specific contexts as well as conditions are not currently being saved.

Minor changes and bug fixes:

- Added command-line options -x, -xe, -xd, -xn, and -xi for controlling the handling of exceptions and events. These command-line options behave just like their command counter-parts.

- The notes window now supports bold, underline, and italics formatting.
- Fixed some zoom and scrolling issues.
- Selecting text in the command, memory, sources, or disassembly windows will now show a light highlight over other instances of the selected text.
- Fixed a bug where interrupting symbol loading would cause symbol loading to fail for the rest of the session.
- NatVis will now reload properly on restarting a session.

1.0.1805.17002

This version includes these updates.

New disassembly window - The disassembly window now includes:

- Scrolling up or down will continuously load more disassembly whenever possible.
- Syntax highlighting for numbers, code addresses, and opcodes.
- Clicking a code symbol will jump the disassembly window to that location.
- Hovering over numbers will show a tooltip that converts that number to other bases.
- Headers signifying the start of a function.

Faster source window - The source window has been updated to be faster and more resource efficient.

Minor changes and bug fixes

- Fixed issues around symbol caching
- Fixed some cases where toggle initial break wasn't usable when the target isn't broken in
- If you hit tab in the command window with nothing available, the cursor will now stay in the input field
- WinDbgNext will now auto-detect bitness when opening CAB files

1.0.1804.18003

This version includes these updates.

Symbol status and cancellation improvements - There are time where the debugger display *BUSY* loading symbols and it's difficult to determine what it's doing and why without !sym noisy enabled. We've updated WinDbg Preview to have some better communication around what it's doing when loading symbols to help troubleshoot any issues. In addition to easily seeing exactly what's happening, we've made some changes that should make cancelling symbols more reliable and the Logs window will contain some of the details that's normally output when !sym noisy is enabled. If you hit View -> Logs you'll get the full noisy symbol loading output without having to turn it on and attempt to reload the symbols.

Experimental notes window - WinDbg Preview now has a window for taking notes. Just hit View -> "Notes" to open it. If you copy/paste into it, DML links will be preserved and still work as if it was the command window. You can also save and load notes files from the "Notes" ribbon when the window is open.

Experimental faster source window - To help improve the performance of WinDbg Preview there us a experimental new source window that is quite a bit more efficient. There's still a few gaps around context menus and syntax highlighting, but we want to give everyone the option of trying it out before it's finished to give us early feedback. Run $UseFastSourceWindow to use it. If you want to go back to the old one, run $UseMonacoSourceWindow. The setting will preserve across sessions, you will need to close and re-open source windows to get the new version.

JSProvider API version 1.2 - For JavaScript extensions that declare support for API version 1.2:

- Any object with a .compareTo method which exits the script will have a custom comparator on it (comparison operators will work in the DX evaluator and elsewhere: e.g.: IModelObject::Compare)

- Any object with a .equals method which exits the script will have a custom equality operator on it (== and != will work in the DX evaluator and elsewhere: e.g.: IModelObject::IsEqualTo)
- Native or data model objects which enter the script will have .compareTo and .equals on them which allow access to any custom comparator or custom equality implementations.

Minor changes and bug fixes:

- .server will now list fully qualified domain name for easier use when there's domain issues around short names.
- Ctrl+G now works in the source window.
- Added address bar to the disassembly window.
- WinDbg Preview will now handle _NT_SYMBOL_PATH in a more expected way.
- Added -server command-line option.
- TTD data model queries can now be displayed progressively, so if you interrupt it you'll still see some results. This feature is still experimental and optional. Run `dx @$cursession.TTD.AsyncQueryEnabled = 1` to enable it.
- The 'dps' command now has links to the source files it refers to.

1.1801.19001.0

This version includes these updates.

Text Highlighting - You can now highlight all instances of selected text directly in the debugger. To use this feature, just select some text in the command window and then click "Highlight" in the command ribbon or hit CTRL+ALT+H. Using one of those on already highlighted text will remove the highlighting.

If you prefer using commands, you can use the "$hl" command:

`$hl ["someValueHere"]` - Highlight give text (or un-highlight if already highlighted)

`$hl clearAll` – Clear all highlighted entries

`$hl caseSensitive [1|0]` - Set highlight matching to case sensitive or case insensitive (default is case insensitive)

This release also includes some minor bug fixes.

1.1712.15003.0

This version includes these updates.

TTD memory queries - You can now query TTD for memory accesses similar to how you query for calls today. This allows you to find all of the reads, writes and execution which access a specific range of memory.

Read and write example: `dx @$cursession.TTD.Memory(startAddress, endAddress, "rw")`

Unique execution example: `dx @$cursession.TTD.Memory(startAddress, endAddress, "ec")`

Settings changes - WinDbg Preview will now automatically save settings between sessions, including your symbol path and source path.

JavaScript Improvements

- 64-bit numbers and numerics in JavaScript now contain a modulo method allowing a true 64-bit modulo operation.
- Objects defined in JavaScript can now implement a custom comparable or equatable notion which will work in dx using standard C++ operators or in LINQ operations. In order to utilize this, the script must declare in the initializeScript array that it supports a new version of the host API by inserting a record "new host.apiVersionSupport(1, 2)". After you've done that you can use these functions in any 'dx' or Data Model Window LINQ query. If the method implements .compareTo(other), it is comparable (comparison operators

work in dx and LINQ). If the method returns a negative value, such as “this < other”. If the method returns zero, “this == other”. If the method returns a positive value “this > other”. If the method implements .equals(other), it is equatable (== works in dx and LINQ). The method must return either true or false.

Minor changes and bug fixes:

- Fixed a bug where the stack and locals windows weren’t working during boot debugging
- Updated the output of LM to more accurately report ProductVersion and similar fields
- Enabled the “step out back” button during TTD sessions
- Added support for -lsrcpath
- The headers in the locals, watch, and model windows now don’t disappear when scrolling down
- When ALT+Tabbing back to WinDbg Preview, the command window will properly preserve cursor location
- Added CTRL+ALT+V shortcut for toggling verbose mode
- You can now disable auto-scrolling of the command window by right-clicking the command window tab and choosing “turn off auto scrolling”
- You can now debug child processes through the launch executable advanced page.

1.0.14.0

This version includes these updates.

Improved process server experience - A new notification in the File menu to show what process server you’re connected to and interacting with has been added. As part of these changes, when ending a debugging session, the process server connection will persist and can be disconnected in the File menu.

New pre-set layout options in the View ribbon - There is a new “Layouts” option in the “View” ribbon. There are currently three layouts: the default, one focused on disassembly, and one minimal.

Time Travel Debugging ribbon - There is an enhanced Time Travel ribbon that will show up when debugging a time travel debugging trace.

Metadata from JavaScript scripts - JavaScript extensions can now return metadata for properties and other constructs. This means that the extension can provide help strings, indicate the display radix for values, and more. Metadata is provided by placing a metadata descriptor on an object via either presence of Symbol.metadataDescriptor or an explicit call to host.metadata.defineMetadata. Function returns, iterated values, and other value contexts can return metadata for their value via host.metadata.valueWithMetadata.

JavaScript API updates - Some potentially source level breaking changes were made to the APIs within the JavaScript provider (including new projected methods and properties on native objects). Existing extensions will not see any of the potentially breaking changes without indicating that they support a new version of the JsProvider API. Support for the new API version is indicated by placing a host.apiVersionSupport record in the array returned by initializeScript with a claim of supporting version 1.1. maybe? .. with a value indicating support for version 1.1.

Changes in API version 1.1 include:

- host.getModuleSymbol and host.getModuleType return null if they cannot find the symbol instead of throwing an exception.
- All native objects have the address property on them in addition to .targetLocation. If the object does not have an address, an exception will be thrown when accessing the property.
- All native objects have new .getObjectValue and .setObjectValue methods on them to access properties on the object which may conflict with names JavaScript places on the object (e.g.: ‘address’) .

Additional JavaScript changes

- JavaScript extensions can now add and remove properties on data model objects via Object.defineProperty and

the delete operator. Adding or registering a JavaScript class as a parent model or type signature is still the strongly preferred way of manipulating the object model.
- JavaScript extensions can now modify global variables within modules in the debug target through a new host.setModuleSymbol API.
- All of the math functions which are on the 64-bit library type (e.g.: .add, .subtract, .multiply, .divide, etc...) are now present on JavaScript numbers as well.
- JavaScript functions and properties can now return values which are enums through custom marshaling. A function or property accessor can return host.typeSystem.marshalAs(value, type...) in order to evoke such custom marshaling.
- The breakpoint command in the script debugger can now break on function names in addition to line/column positions.
- Type objects in JavaScript extensions have access to their containing module through the .containingModule property.

Minor changes and bug fixes:

- Fixed formatting of conditional ribbon tabs to be less confusing.
- Re-worked DML to be stricter in parsing to improve performance.
- Various fixes with the performance and behavior of CTRL+F.
- Added a warning when running un-elevated prior to trying to use TTD.
- Added the option to override automatic target bitness detection.
- Disabled various file menu and ribbon options when they can't be used (like "Go" when in a dump file).

Known issues:

- SOS will not work on x86 traces.

1.0.13.0

This version adds Time Travel Tracing. Time Travel Debugging, allows you to record a process, then replay it later both forwards and backwards. Time Travel Debugging (TTD) can help you debug issues easier by letting you "rewind" your debugger session, instead of having to reproduce the issue until you find the bug. For more information, see Time Travel Debugging - Overview.

1.0.12.0

This version was the first release of WinDbg Preview. For general information on the features available in WinDbg Preview, Debugging Using WinDbg Preview.

See Also

Debugging Using WinDbg Preview

WinDbg Preview - Installation

10/7/2019 • 2 minutes to read • Edit Online

Installation

This section describes how to install the WinDbg Preview debugger.

The WinDbg Preview debugger is available in the Microsoft Store. It requires Windows 10 Anniversary Update to install. To install it, open the Microsoft Store and search for "WinDbg Preview", or click here.

Once the app is a located, click on it to download and install.

Checking for updates

1. Open the Store app and click on your account picture next to the search box.
2. Click on **Downloads and Updates** to check for updates.
3. On the downloads and updates page, select **Get updates**.

Debugger coexistence

The WinDbg preview coexists with the classic WinDbg debugger on the same machine, so you can work with both versions at the same time.

See Also

Debugging Using WinDbg Preview

WinDbg Preview - Command line startup options

10/14/2019 • 3 minutes to read • Edit Online

Starting WinDbg Preview

After WinDbg Preview is installed, WinDbgX.exe is available to run from any directory location.

Command line startup options

```
WinDbgX [options]
```

This following tables summarizes the available command line options.

General Options

OPTION	DESCRIPTION
-c "*command*"	Executes a command line after the debugger is attached. This command must be enclosed in quotation marks. Multiple commands can be separated with semicolons.
-v	Enables verbose output in the debugger.
-T *Title*	Sets the window title.
-logo *LogFile*	Log Open. Begins logging information to a log file. If the file exists, it will be overwritten.
-loga *LogFile*	Log Append. Begins logging information to a log file. If the file exists, it will be appended to.
-e *EventHandle*	Signals the event with the given handle after the next exception in a target.
-?	Displays a summary of commands available.

Kernel Options

OPTION	DESCRIPTION
-k [ConnectType]	Starts a kernel debugging session. If **-k** is used without any *ConnectType* options following it, it must be the final entry on the command line.
-kqm	Starts KD in quiet mode.
-kl	Starts a kernel debugging session on the same machine as the debugger.
-kx *ExdiOptions*	Starts a kernel debugging session using an EXDI driver.

OPTION	DESCRIPTION
-d	After a reboot, the debugger will break into the target computer as soon as a kernel module is loaded.

User Mode Options

OPTION	DESCRIPTION
-o	Debugs all processes launched by the target application (child processes).
-g	Ignores the initial breakpoint in target application.
-G	Ignores the final breakpoint in target application.
-pv	Specifies that the debugger should attach to the target process noninvasively.
-hd	Specifies that the debug heap should not be used.
-cimp	Specifies that any processes created will use an implicit command-line set by the server instead of a user-given command-line string from the client.

Target Options

OPTION	DESCRIPTION
-remote *ClientTransport*	Connects to a debugging server that is already running. For an explanation of the possible *ClientTransport* values, see Activating a Debugging Client. When this parameter is used, it must be the first parameters on the command line.
-server *ServerTransport*	Creates a debugging server that can be accessed by other debuggers. For an explanation of the possible *ServerTransport* values, see Activating a Debugging Server.
-premote *SmartClientTransport*	Creates a smart client, and connects to a process server that is already running. For an explanation of the possible SmartClientTransport values, see Activating a Smart Client.
-p *PID*	Specifies the decimal process ID to be debugged.
-tid *TID*	Specifies the thread ID of a thread to be resumed when the debugging session is started.
-psn *ServiceName*	Specifies the name of the service contained in the process to be debugged. This is used to debug a process that is already running.
-pn *ProcessName*	Specifies the name of the process to be debugged.

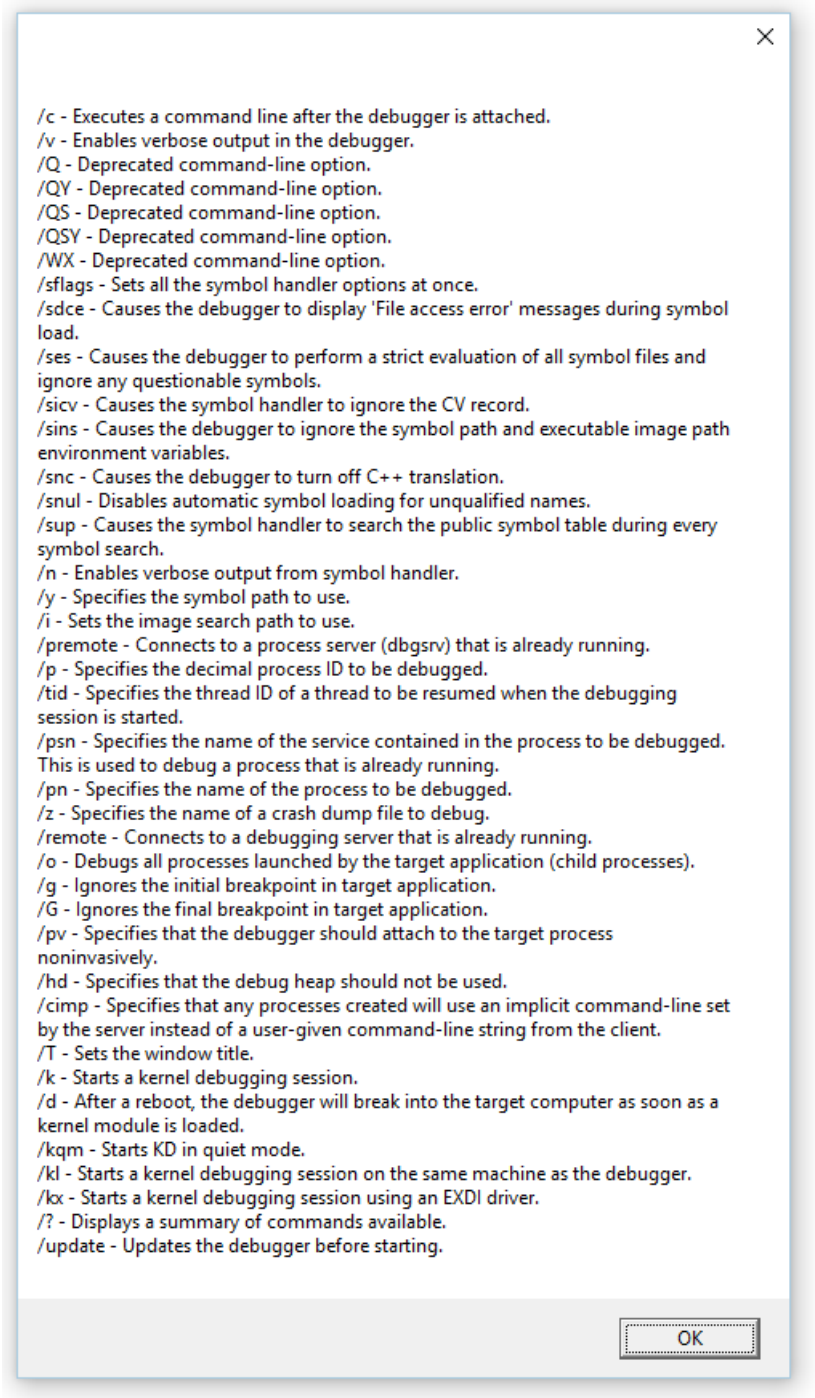
OPTION	DESCRIPTION
-z *DumpFile*	Specifies the name of a crash dump file to debug. If the path and file name contain spaces, this must be surrounded by quotation marks.
-openPrivateDumpByHandle *Handle*	*Microsoft internal use only*. Specifies the handle of a crash dump file to debug.
-debugArch x86 -or- amd64	Override the autodetect behavior and set the target bitness for the debugger.
executable	Specifies the command line of an executable process. This is used to launch a new process and debug it. This has to be the final item on the command line. All text after the executable name is passed to the executable as its argument string.

Symbol Options

OPTION	DESCRIPTION
-y *SymbolPath*	Specifies the symbol path to use. Separate multiple paths with a semicolon (;). If the path contains spaces, it should be enclosed in quotation marks. For details, and for other ways to change this path, see Symbol Path.
-n	Noisy symbol load. Enables verbose output from symbol handler.
-i *ImagePath*	Sets the image search path to use.
-sdce	Causes the debugger to display 'File access error' messages during symbol load.
-ses	Causes the debugger to perform a strict evaluation of all symbol files and ignore any questionable symbols.
-sicv	Causes the symbol handler to ignore the CV record
-sins	Causes the debugger to ignore the symbol path and executable image path environment variables.
-snc	Causes the debugger to turn off C++ translation.
-snul	Disables automatic symbol loading for unqualified names.
-sup	Causes the symbol handler to search the public symbol table during every symbol search
-sflags	Sets all the symbol handler options at once.

For general information on the startup parameters, see WinDbg Command-Line Options.

You can use /? to list the supported command line options.



See Also

Debugging Using WinDbg Preview

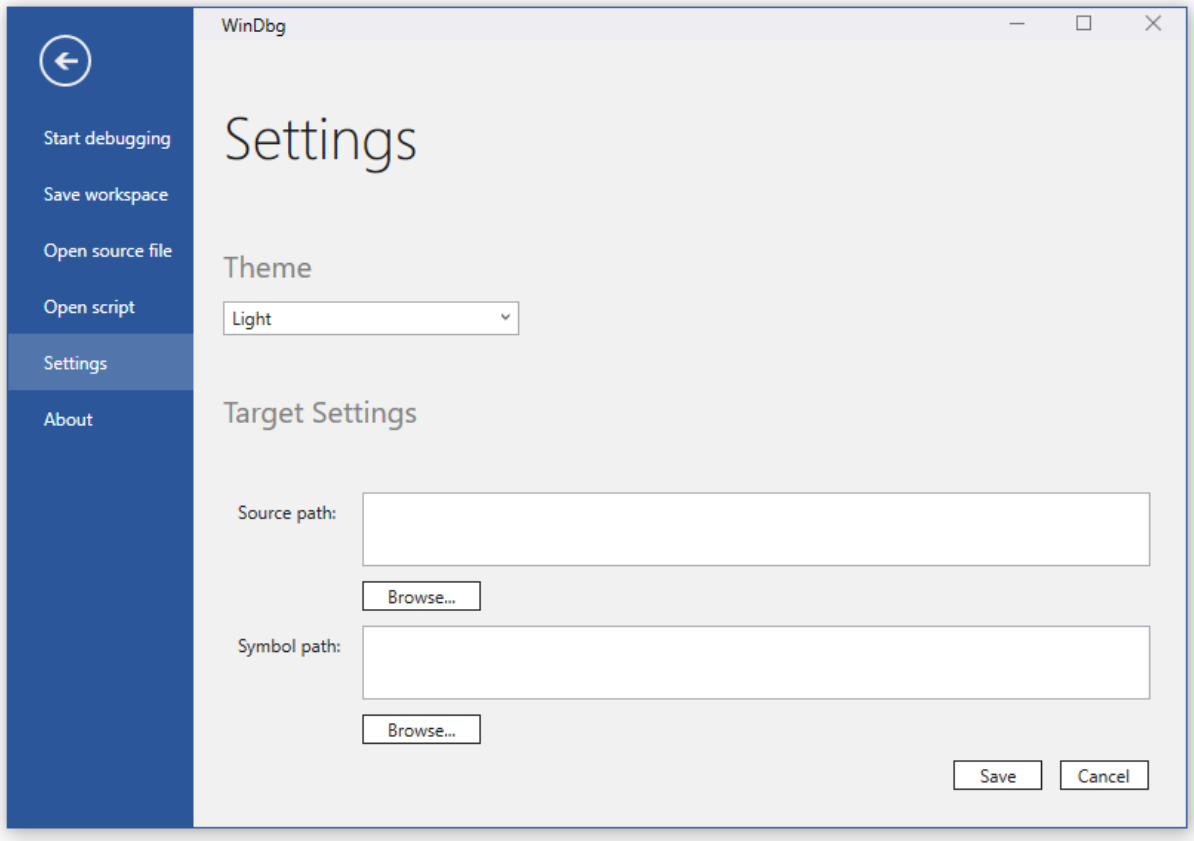
WinDbg Preview - Settings and workspaces

12/5/2018 • 3 minutes to read • Edit Online

This section describes how to setup and configure the WinDbg Preview debugger.

Settings

Use the settings menu to set the source and symbol path as well as choose the light and dark theme for the debugger.



For more information on setting the paths, see Accessing Symbols for Debugging and Source Code Debugging in WinDbg.

Workspaces

Workspaces allows you to save configuration information in the target connection information file.

The options in workspaces are saved upon closing the debugger or can be manually saved using *File -> Save Workspace*.

Workspaces are automatically loaded when launching from the recent targets list or they can be manually loaded in the file menu.

In addition to the target connection information, the following settings are stored in the workspaces file.

General Settings

> **NOTE**
> This list and format isn't final and is subject to change.

SETTING	DEFAULT	DESCRIPTION
FinalBreak	true	If true, ignores the final breakpoint (-g command-line option).
SourceDebugging	true	Toggles between source or assembly mode.
DebugChildProcesses	false	(User mode only) If true will debug child processes launched by the target application. (-o command-line option).
Noninvasive	false	Specifies non-invasive attach (-pv command-line option).
NoDebugHeap	false	Specifies the debug heap should not be used (-hd command-line option).
Verbose	false	When verbose mode is turned on, some display commands (such as register dumping) produce more detailed output. (-v command-line option).
Elevate	-	Used internally by WinDbg - Do not modify.
Restartable	-	Used internally by WinDbg - Do not modify.
UseImplicitCommandLine	false	Use implicit command-line (-cimp command-line option). This starts the debugger with an implicit command line instead of an explicit process to run.

For more information about the command line options, see WinDbg Command-Line Options.

Symbol Settings

SETTING	DEFAULT	DESCRIPTION
SymbolOptionsOverride	0	An explicit symbol option mask, in the form of a single hex number.
ShouldOverrideSymbolOptions	false	If set to *true* override all of the symbol options listed below with the provided symbol option mask, described above.
SymOptExactSymbols	false	This option causes the debugger to perform a strict evaluation of all symbol files.

SETTING	DEFAULT	DESCRIPTION
SymOptFailCriticalErrors	false	This symbol option causes file access error dialog boxes to be suppressed.
SymOptIgnoreCvRec	false	This option causes the symbol handler to ignore the CV record in the loaded image header when searching for symbols.
SymOptIgnoreNtSympath	false	This option causes the debugger to ignore the environment variable settings for the symbol path and the executable image path.
SymOptNoCpp	false	This symbol option turns off C++ translation. When this symbol option is set, :: is replaced by __ in all symbols.
SymOptNoUnqualifiedLoads	false	This symbol option disables the symbol handler's automatic loading of modules. When this option is set and the debugger attempts to match a symbol, it will only search modules which have already been loaded.
SymOptAutoPublics	false	This symbol option causes DbgHelp to search the public symbol table in a .pdb file only as a last resort. If any matches are found when searching the private symbol data, the public symbols will not be searched. This improves symbol search speed.
SymOptDebug	false	This symbol option turns on noisy symbol loading. This instructs the debugger to display information about its search for symbols.

For more information on symbol options, see Symbol Options.

Window layout settings

Window layout is saved globally and are not saved in the workspaces file.

Workspaces XML file

The workspace and target connection information is stored in XML format.

The following file, shows an example workspaces configuration file.

```
<?xml version="1.0" encoding="utf-8"?>
<TargetConfig Name="C:\paint.dmp" LastUsed="2017-08-03T21:34:20.1013837Z">
  <EngineConfig />
  <EngineOptions>
    <Property name="FinalBreak" value="true" />
    <Property name="SourceDebugging" value="true" />
    <Property name="DebugChildProcesses" value="false" />
    <Property name="Noninvasive" value="false" />
    <Property name="NoDebugHeap" value="false" />
    <Property name="Verbose" value="false" />
    <Property name="SymbolOptionsOverride" value="0" />
    <Property name="ShouldOverrideSymbolOptions" value="false" />
    <Property name="SymOptExactSymbols" value="false" />
    <Property name="SymOptFailCriticalErrors" value="false" />
    <Property name="SymOptIgnoreCvRec" value="false" />
    <Property name="SymOptIgnoreNtSympath" value="false" />
    <Property name="SymOptNoCpp" value="false" />
    <Property name="SymOptNoUnqualifiedLoads" value="false" />
    <Property name="SymOptAutoPublics" value="false" />
    <Property name="SymOptDebug" value="false" />
    <Property name="Elevate" value="false" />
    <Property name="Restartable" value="true" />
    <Property name="UseImplicitCommandLine" value="false" />
  </EngineOptions>
  <TargetOptions>
    <Option name="OpenDump">
      <Property name="DumpPath" value="C:\paint.dmp" />
    </Option>
  </TargetOptions>
</TargetConfig>
```

Note that this file format will continue to evolve as more features are added to the WinDbg Preview debugger.

See Also

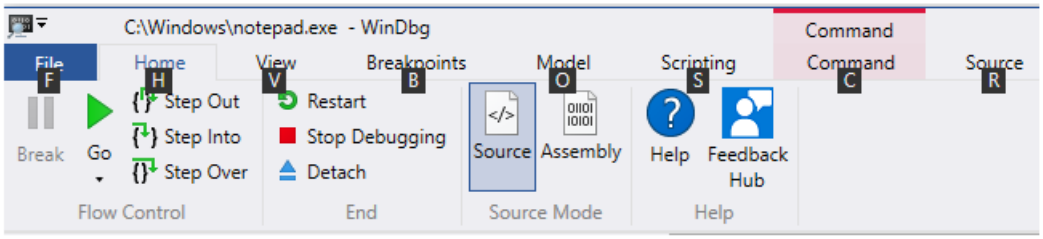
Debugging Using WinDbg Preview

WinDbg Preview keyboard shortcuts

12/5/2018 • 2 minutes to read • Edit Online

This section summarizes the keyboard shortcuts for the WinDbg preview debugger.

All of the ribbon menu options are available using the **Alt +** the first letter of the option. For example **Alt + H** to go to the home menu, **Alt + H + P** to view help.



Flow control

KEYSTROKE	DESCRIPTION
F5	Continue
F10	Step over
F11	Step Into
Shift+F11	Step out
F7	Run to line
Ctrl+Shift+I	Set instruction pointer to highlighted line
Ctrl+Break or Alt+Del	Break
Ctrl+Shift+F5	Restart
Shift+F5	Stop debugging
Alt+H,D	Detach

Setup

KEYSTROKE	DESCRIPTION
F6	Attach to process
Ctrl+R	Connect to remote
Ctrl+D	Open dump file
Ctrl+K	Attach to kernel debugger

KEYSTROKE	DESCRIPTION
Ctrl+E	Launch process
Ctrl+P	Launch app package

Breakpoints

KEYSTROKE	DESCRIPTION
F9	Toggle breakpoint on highlighted line
Ctrl+Alt+K	Toggle initial break
Alt+B,A	Add breakpoint

Windowing

KEYSTROKE	DESCRIPTION
Ctrl+Tab	Open window changer
Ctrl+1	Open/focus on command window
Ctrl+2	Open/focus on watch window
Ctrl+3	Open/focus on locals window
Ctrl+4	Open/focus on registers window
Ctrl+5	Open/focus on memory window
Ctrl+6	Open/focus on stack window
Ctrl+7	Open/focus on disassembly window
Ctrl+8	Open/focus on breakpoints window
Ctrl+9	Open/focus on thread window

Scripting

KEYSTROKE	DESCRIPTION
Ctrl+Shift+O	Open script
Ctrl+Shift+Enter	Execute script
Ctrl+S	Save script
Alt+S,N	New script

KEYSTROKE	DESCRIPTION
Alt+S,U	Unlink script

Stack navigation

KEYSTROKE	DESCRIPTION
Ctrl+↑ / ↓	Move up/down a stack frame

Help

KEYSTROKE	DESCRIPTION
F1	Open help file
Shift+F1	Search selection online (source window)

Misc.

KEYSTROKE	DESCRIPTION
Ctrl+Alt+V	Toggle Verbose Mode

See Also

Debugging Using WinDbg Preview

WinDbg Preview - Start a user mode session

12/5/2018 • 2 minutes to read • Edit Online

This section describes how to start a user mode session with the WinDbg preview debugger.

Select *File*, *Start debugging*, and select either of these four options:

- *Launch Executable* - Starts an executable and attaches to it by browsing for the target.
- *Launch Executable (advanced)* - Starts an executable and attaches to it using a set of dialog boxes with advanced options.
- *Attach to a process* - Attaches to an existing process.
- *Launch App Package* - Launches and attaches to an app package.

All four options are described here.

Launch Executable

Use this option to starts an executable and attach to it.

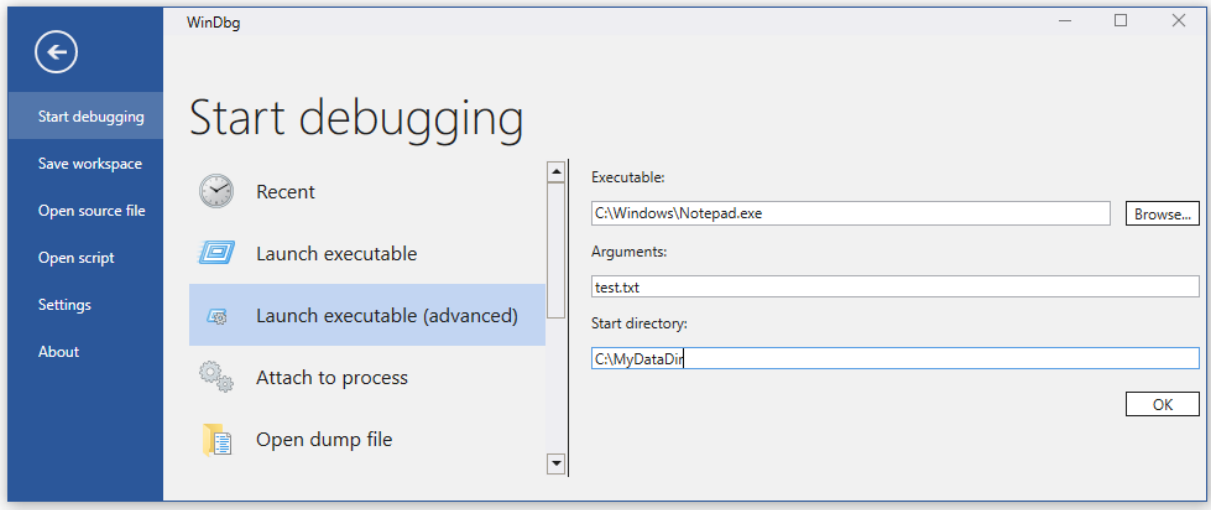
Browse to the desired executable in the provided file dialog and open it.

Launch Executable (advanced)

Use this option to start an executable and attach to it using a set of text boxes with advanced options.

Specify the following options:

- Path to the executable, such as *C:\Windows\notepad.exe*
- Optional arguments to provide to the executable when launched
- Optional start directory location



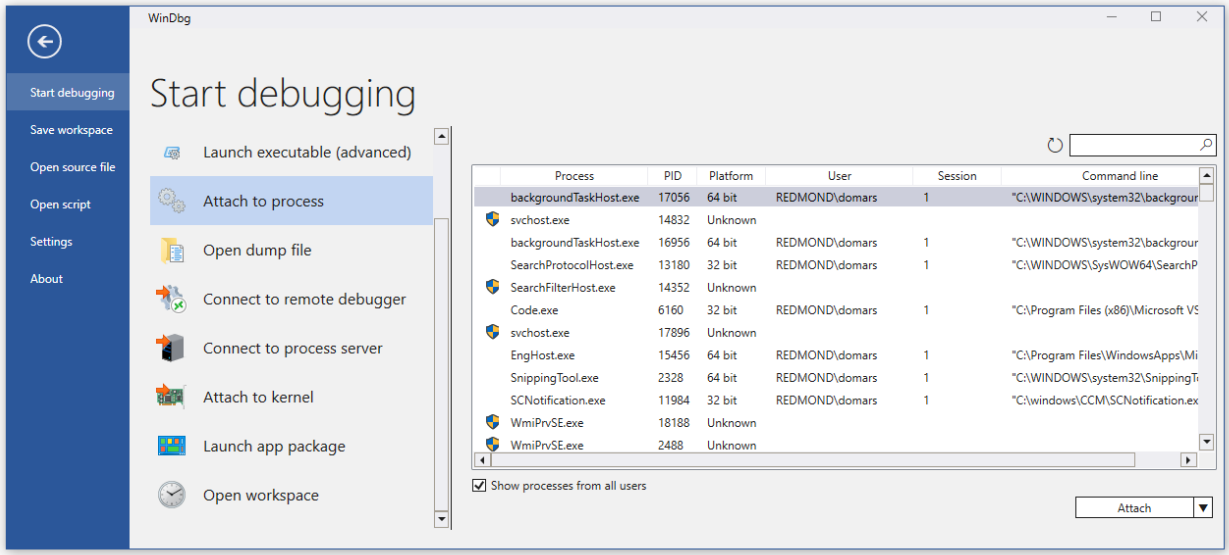
Attach to a process

Use this option to attach to an existing process.

Select *Show process from all users* to show additional processes.

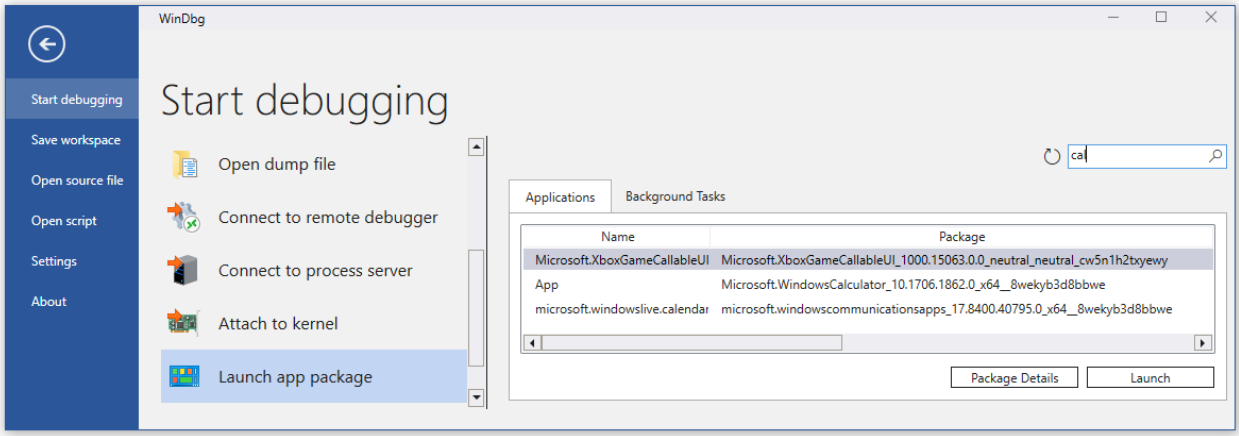
> **NOTE**
>
> Items with a UAC shield will need the debugger to be elevated to attach.

Use the pull down dialog on the *Attach* button to select *non-invasive attach*.



Launch App Package

Use this option to launch and attach to an app package. Use the search box to locate a specific app or background task. Use the Package Details button to display information about the package.



See Also

[Debugging Using WinDbg Preview]

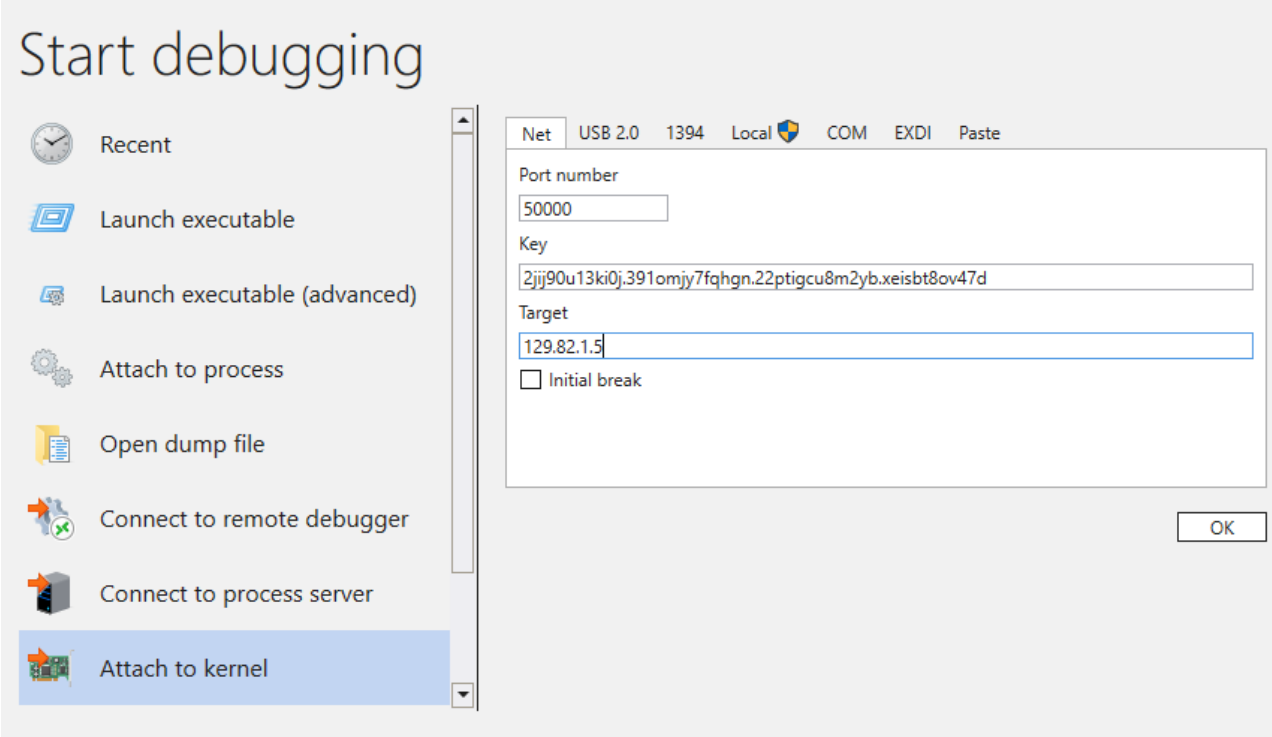
WinDbg Preview - Starting a kernel mode session

12/5/2018 • 2 minutes to read • Edit Online

This topic describes how to start a kernel mode session with WinDbg Preview.

The process is very similar to how it has been done with previous versions of WinDbg. Select the tab for the type of transport you're using, fill in the required fields, and click OK.

> **NOTE**
> Local kernel debugging requires WinDbg Preview to be launched elevated.



The *Paste* tab allows you to paste in custom connection strings.

If you are not familiar with setting up a debugger kernel mode session, see Getting Started with WinDbg (Kernel-Mode)

See Also

Debugging Using WinDbg Preview

WinDbg Preview - File Menu

12/5/2018 • 2 minutes to read • Edit Online

This topic describes how to how to use the file menu.

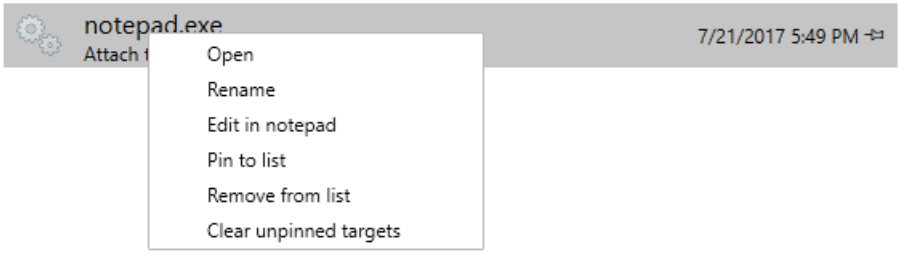
Start debugging

When you first open the file menu, you'll see *Start debugging* and your recent debugger targets. Use *Start debugging* to configure new and open previous debugger sessions.

Recent

The recent list contains a list of your recent workspaces and debugger connections. For more information on working settings an workspaces see WinDbg Preview Setup – Settings and workspaces.

You can use the right click menu to manage your workspaces, like pinning, renaming and moving them. As well as editing them in notepad.



Start a new session

Use the other tabs in the *Start debugging* section to start a new debugger session, like attaching or launching a process. For more information on starting a new session see WinDbg Preview - Start a user-mode session and WinDbg Preview - Start a kernel mode session

Save workspace

Use *Save workspace* to save the current workspace.

Session connection information is stored in workspace configuration files. Workspace files are stored with a .debugTarget file extension.

The default location for workspace files is:

```
C:\Users\*UserName*\AppData\Local\DBG\targets
```

Open source file

Use *Open source file* to open a source file. Do this when you want to work with additional source files that have not been loaded because of code execution. For more information on working with source files, see Source Code Debugging in WinDbg

Open script

Use *Open script* to open an existing Javascript or NatVis script. For more information on working with scripts see WinDbg Preview - Scripting Menu.

Settings

Use the settings menu to set the source and symbol path as well as choose the light and dark theme for the

debugger. For more information on settings see WinDbg Preview Setup – settings and workspaces.

About

Use *About* to display build version information for the debugger. You can use also use this screen to view the Microsoft privacy statement.

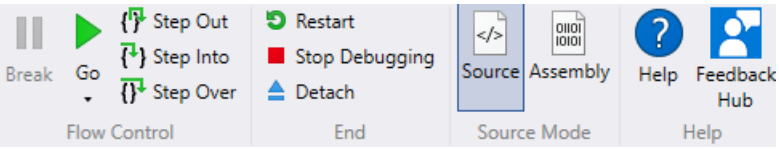
See Also

Debugging Using WinDbg Preview

WinDbg Preview - Home Menu

12/5/2018 • 2 minutes to read • Edit Online

This topic describes how to work with the home menu.



Flow Control

Use the *Flow Control* buttons to break into a connected debugging target, resume code execution on the target and step in to and out of code.

End

Use the *End* buttons to restart, detach and stop debugging.

Source Mode

Use *Source Mode* buttons to toggle between source code and assembly views.

Help (Support)

Use *Help* buttons to do the following:

- Review Help
- Send Feedback (For more information on sending feedback to improve WinDbg, see Providing feedback.)

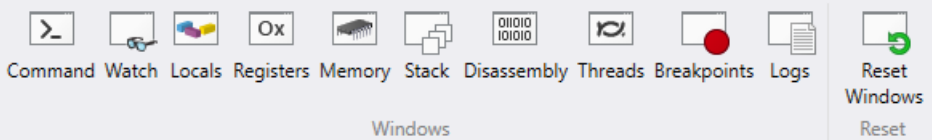
See Also

Debugging Using WinDbg Preview

WinDbg Preview - View Menu

12/5/2018 • 2 minutes to read • Edit Online

This section describes how work with the view menu in WinDbg Preview.



The view menu will open a new Window for each item, or bring focus to the existing Window, if one is already open.

Command

The command Window allows you to enter debugger commands. For more information about debugger commands, see Debugger Commands.

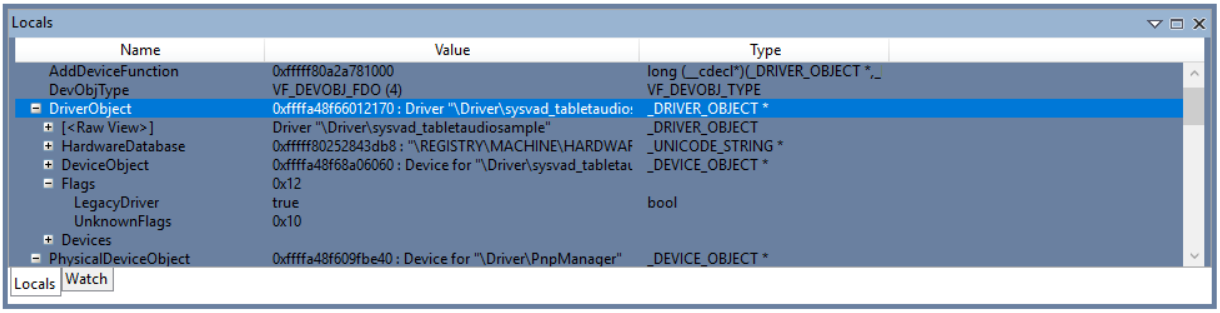
Watch

The watch Window allows you to watch local variables and registers.

The locals and watch windows are both based off of the data model that is used by the dx command. This means the locals and watch windows will benefit from any NatVis or JavaScript extensions you have loaded, and support full LINQ queries, just like the dx command. For more information about the data model, see WinDbg Preview - Data Model.

Locals

The locals window displays information about all of the local variables in the current scope. The locals window will highlight values that have changed during the previous code execution.



Registers

Registers displays the contents of the processors registers when they are available. For more information about registers, see Registers and Viewing and Editing Registers in WinDbg.

Memory

Use the memory window to display memory locations. In addition to providing a memory address, you can use the Pseudo-Register values such as $scopeip and $eventip to examine memory. Pre-append the @ symbol to use

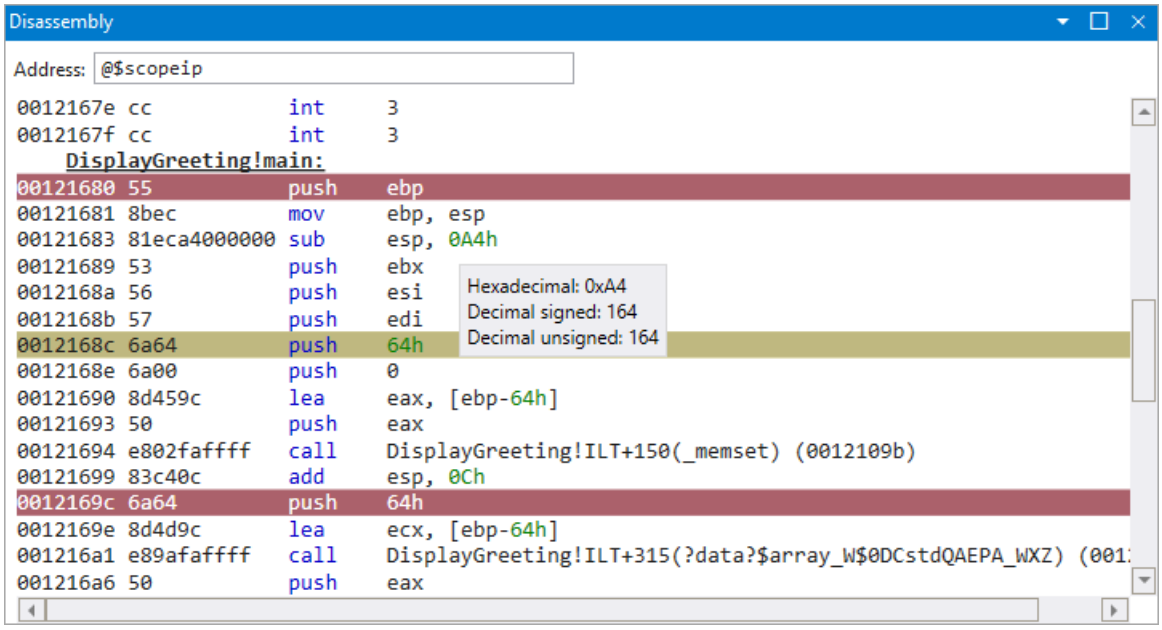
the pseudo-register values in the memory window, for example, `@$scopeip`. For more information, see Pseudo-Register Syntax

Stack

Use the stack Window to view the current call stack. The stack window provides basic highlighting of the current frame.

Disassembly

The disassembly window highlights the current instruction and retains that position when you scroll.

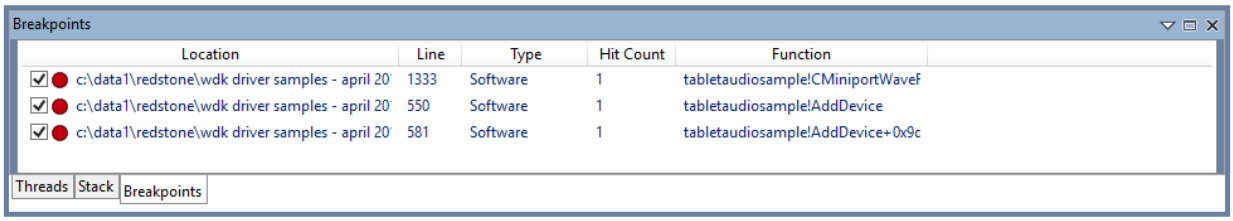


Threads

The thread window highlights the current thread.

Breakpoints

Use the breakpoints window to view, enable and clear breakpoints.



Logs

This log is of the WinDbg Preview internals. It can be viewed to monitor long running processes and for troubleshooting the debugger itself.

You can continue to create a traditional debugger command log, using the .logopen command. For more information on that, see Keeping a Log File in WinDbg.

Reset Windows

Use this function to reset the debugger windows to their default positions.

See Also

Debugging Using WinDbg Preview

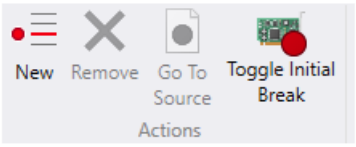
WinDbg Preview - Breakpoints

12/5/2018 • 2 minutes to read • Edit Online

This section describes how to work with breakpoints using the WinDbg preview debugger.

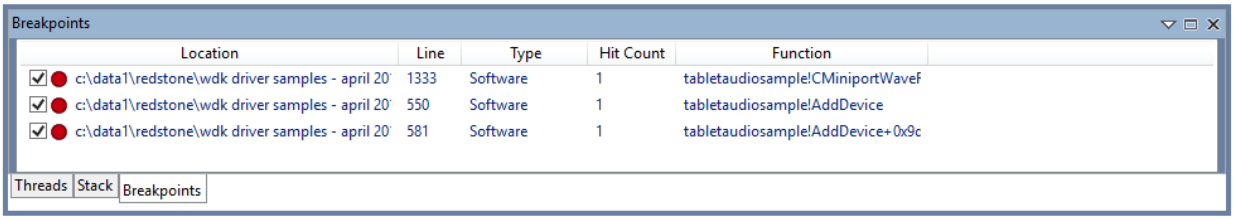
Breakpoints Menu

Use the breakpoints menu to create new and remove existing breakpoints as well as toggle the initial breakpoint (initial breakpoint is currently kernel only).



Breakpoints Window

Use the breakpoints window, opened via the View menu, to view, set and clear breakpoints. You can also double-click a breakpoint to open its source file.



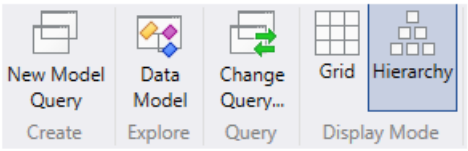
The breakpoint window keep a running total of each time the breakpoint is hit.

The general process of working with breakpoints is similar to previous versions of WinDbg. For more information about setting breakpoints, see Setting Breakpoints in WinDbg.

WinDbg Preview - Data Model

12/5/2018 • 2 minutes to read • Edit Online

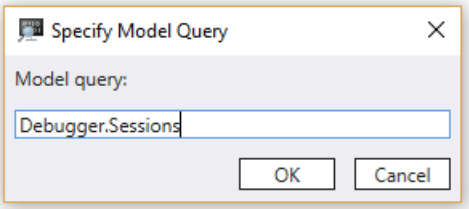
This section describes how to work with the data model menu in the WinDbg Preview debugger.



New Model Query

Use the New Model Query dialog to create a new model query. You can put anything here you'd put into a normal `dx` command.

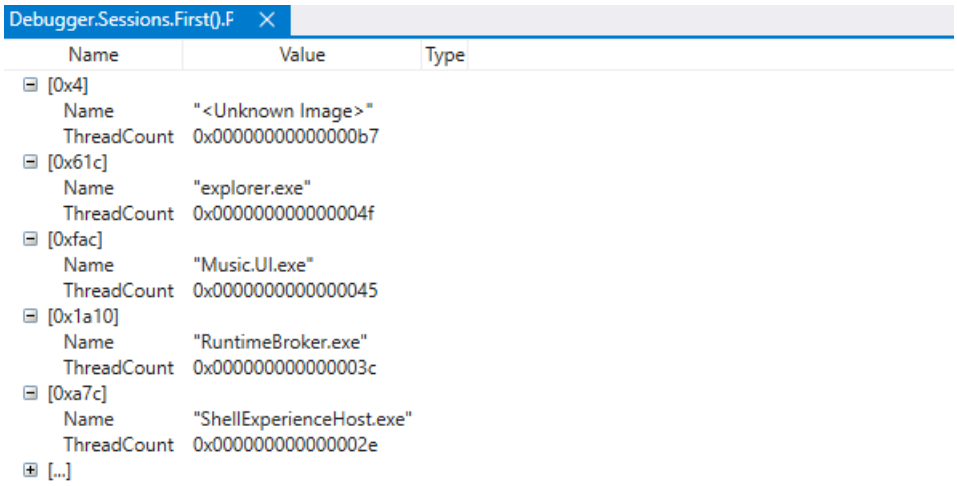
For example, specify `Debugger.Sessions` to examine the debugger sessions objects.



For general information about the debugger objects refer to dx (Display Debugger Object Model Expression).

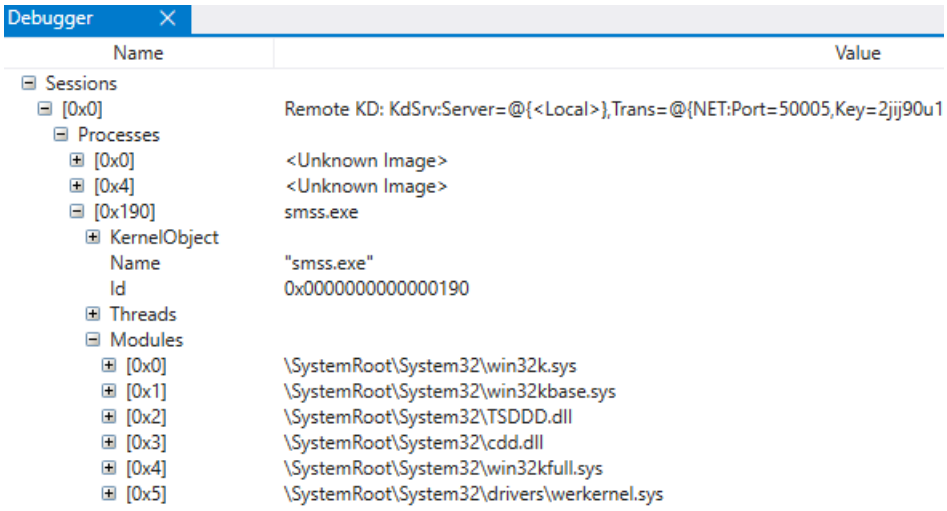
Use LINQ queries to dig deeper into the session. This query shows the top 5 processes running the most threads.

```
Debugger.Sessions.First().Processes.Select(p => new { Name = p.Name, ThreadCount = p.Threads.Count()
}).OrderByDescending(p => p.ThreadCount),5
```



Data Model Explorer

Use the data model explorer to quickly browse every data model object in the `Debugger` namespace.



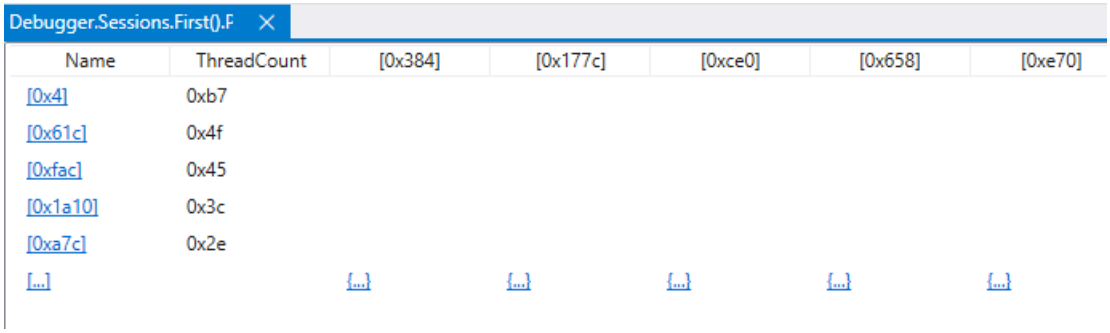
Change Query

Use change query to change the query that is used in the active data model window.

Display Mode

Use display mode to toggle between grid and hierarchy display mode. You can right-click column headers to hide or show more columns.

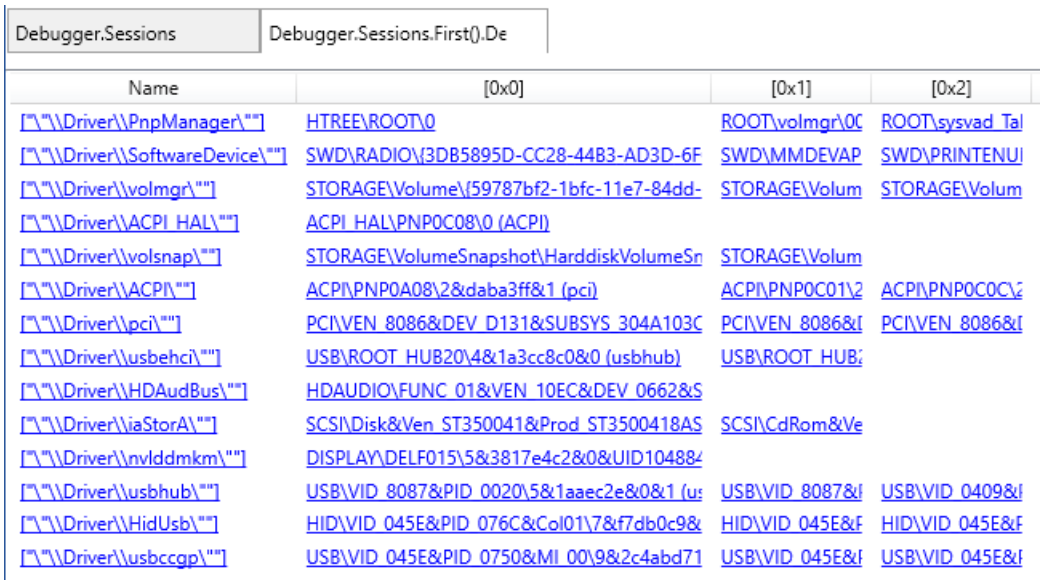
Grid mode can be useful to dig down in the objects. For example, here is the previous top threads query in grid view.



When you click on any underlined item a new tab is opened and a query is run to display that information.

This query shows the devices in the plug and play device tree grouped by the name of the physical device object's driver.

```
Debugger.Sessions.First().Devices.DeviceTree.Flatten(n => n.Children).GroupBy(n => n.PhysicalDeviceObject->Driver->DriverName.ToDisplayString())
```



Debugger.Sessions | Debugger.Sessions.First().De

Name	[0x0]	[0x1]	[0x2]
["\"\\Driver\\PnpManager\""]	HTREE\ROOT\0	ROOT\volmgr\0(	ROOT\sysvad_Tal
["\"\\Driver\\SoftwareDevice\""]	SWD\RADIO\{3DB5895D-CC28-44B3-AD3D-6F	SWD\MMDEVAP	SWD\PRINTENUI
["\"\\Driver\\volmgr\""]	STORAGE\Volume\{59787bf2-1bfc-11e7-84dd-	STORAGE\Volum	STORAGE\Volum
["\"\\Driver\\ACPI_HAL\""]	ACPI_HAL\PNP0C08\0 (ACPI)		
["\"\\Driver\\volsnap\""]	STORAGE\VolumeSnapshot\HarddiskVolumeSn	STORAGE\Volum	
["\"\\Driver\\ACPI\""]	ACPI\PNP0A08\2&daba3ff&1 (pci)	ACPI\PNP0C01\2	ACPI\PNP0C0C\2
["\"\\Driver\\pci\""]	PCI\VEN_8086&DEV_D131&SUBSYS_304A103C	PCI\VEN_8086&[	PCI\VEN_8086&[
["\"\\Driver\\usbehci\""]	USB\ROOT_HUB20\4&1a3cc8c0&0 (usbhub)	USB\ROOT_HUB2	
["\"\\Driver\\HDAudBus\""]	HDAUDIO\FUNC_01&VEN_10EC&DEV_0662&S		
["\"\\Driver\\iaStorA\""]	SCSI\Disk&Ven_ST350041&Prod_ST3500418AS	SCSI\CdRom&Ve	
["\"\\Driver\\nvlddmkm\""]	DISPLAY\DELF015\5&3817e4c2&0&UID10488		
["\"\\Driver\\usbhub\""]	USB\VID_8087&PID_0020\5&1aaec2e&0&1 (us	USB\VID_8087&[	USB\VID_0409&[
["\"\\Driver\\HidUsb\""]	HID\VID_045E&PID_076C&Col01\7&f7db0c9&	HID\VID_045E&F	HID\VID_045E&F
["\"\\Driver\\usbccgp\""]	USB\VID_045E&PID_0750&MI_00\9&2c4abd71	USB\VID_045E&[	USB\VID_045E&[

See Also

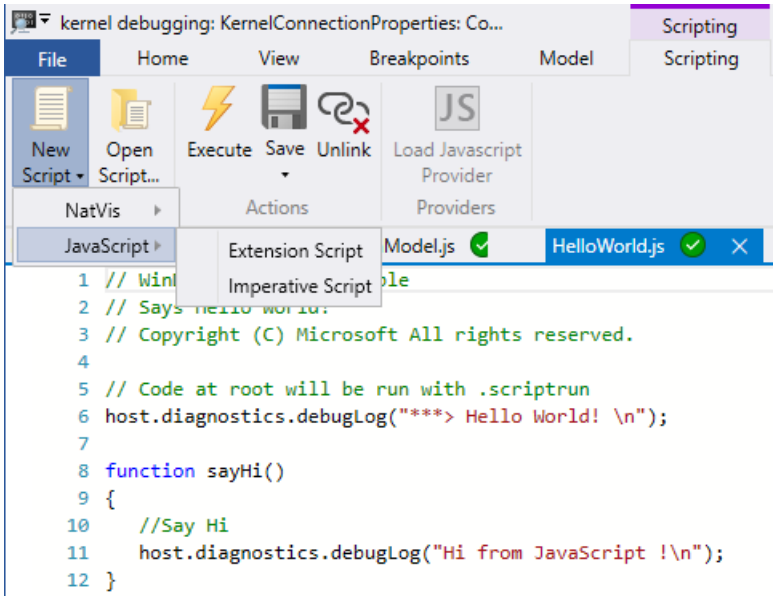
dx (Display Debugger Object Model Expression)

Debugging Using WinDbg Preview

WinDbg Preview - Scripting

10/25/2019 • 2 minutes to read • Edit Online

This section describes how to use the scripting support in the WinDbg Preview.



The WinDbg Preview script window features basic syntax highlighting, IntelliSense, and error recognition.

Use the ribbon buttons to:

- Create a new script
- Open an existing script
- Execute a script
- Save a script
- Unlink a script
- Load the JavaScript Provider

You can also automatically execute scripts by right-clicking in the script window and selecting *Execute Script on Save*. When you successfully load a script, a green check box will appear on the script title bar. If there are errors in the script, a red x will be displayed.

JavaScript Scripting

To start using JavaScript, you must first be debugging a target. When you're ready to start working on your JavaScript, click "Load JavaScript Provider". After that you can create a new JavaScript, by picking between these two types of script templates.

- **Extension Script** - A script which is designed to act as an extension to the debugger. It manipulates the object model of the debugger and provides continued functionality. No action happens upon hitting the *Execute* button on the ribbon.
- **Imperative Script** - A script which is designed to perform an action each and every time the *Execute* button is clicked on the ribbon. Such a script does not generally modify the object model of the debugger.

For more information about working with JavaScript, see these topics:

JavaScript Debugger Scripting

Native Debugger Objects in JavaScript Extensions

JavaScript Debugger Example Scripts

NatVis Scripting

Use **New Script** > **NatVis** to open the following blank NatVis template.

```
<AutoVisualizer xmlns="https://schemas.microsoft.com/vstudio/debugger/natvis/2010">
  <Type Name="">
  </Type>
</AutoVisualizer>
```

For more information about working with NatVis, see Debugger Objects in NatVis.

See Also

Debugging Using WinDbg Preview

Time Travel Debugging - Overview

10/25/2019 • 7 minutes to read • Edit Online

What is Time Travel Debugging?

Time Travel Debugging, is a tool that allows you to record an execution of your process running, then replay it later both forwards and backwards. Time Travel Debugging (TTD) can help you debug issues easier by letting you "rewind" your debugger session, instead of having to reproduce the issue until you find the bug.

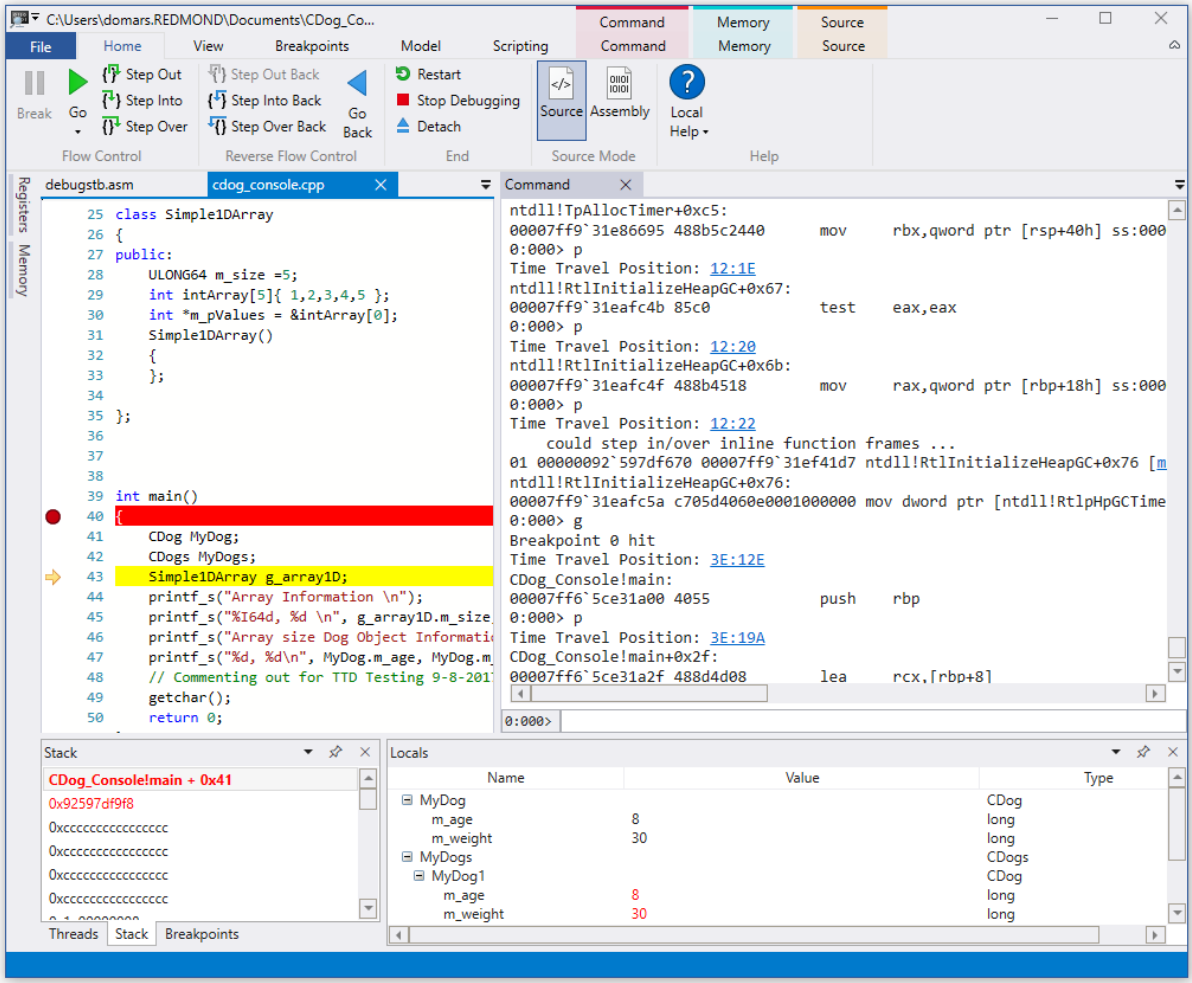
TTD allows you to go back in time to better understand the conditions that lead up to the bug and replay it multiple times to learn how best to fix the problem.

TTD can have advantages over crash dump files, which often are missing the code execution that led up to the ultimate failure.

In the event you can't figure out the issue yourself, you can share the trace with a co-worker and they can look at exactly what you're looking at. This can allow for easier collaboration than live debugging, as the recorded instructions are the same, where the address locations and code execution will be different on different PCs. You can also share a specific point in time to help your co-worker figure out where to start.

TTD is efficient and works to add as little as possible overhead as it captures code execution in trace files.

TTD includes a set of debugger data model objects to allow you to query the trace using LINQ. For example, you can use TTD objects to locate when a specific code module was loaded or locate all of the exceptions.



Comparison of Debugging Tools

This table summarizes the pros and cons of the different debugging solutions available.

APPROACH	PROS	CONS
Live debugging	Interactive experience, sees flow of execution, can change target state, familiar tool in familiar setting.	Disrupts the user experience, may require effort to reproduce the issue repeatedly, may impact security, not always an option on production systems. With repro difficult to work back from point of failure to determine cause.
Dumps	No coding upfront, low-intrusiveness, based on triggers.	Successive snapshot or live dumps provide a simple “over time” view. Overhead is essentially zero if not used.
Telemetry & logs	Lightweight, often tied to business scenarios / user actions, machine learning friendly.	Issues arise in unexpected code paths (with no telemetry). Lack of data depth, statically compiled into the code.
Time Travel Debugging (TTD)	Great at complex bugs, no coding upfront, offline repeatable debugging, analysis friendly, captures everything.	Large overhead at record time. May collect more data that is needed. Data files can become large.

TTD Availability

TTD is available on Windows 10 after installing the WinDbg Preview app from the Store. WinDbg Preview is an improved version of WinDbg with more modern visuals, faster windows, a full-fledged scripting experience, with built in support for the extensible debugger data model. For more information on downloading WinDbg Preview from the store, see Debugging Using WinDbg Preview.

Administrator rights required to use TTD

To use TTD, you need to run the debugger elevated. Install WinDbg Preview using an account that has administrator privileges and use that account when recording in the debugger. In order to run the debugger elevated, right-click on the WinDbg Preview icon in the Start menu and then select More > Run as Administrator.

Video Training

To learn more about TTD see these videos.

Defrag Tools 185 - Ivette and JamesP go over the basics of TTD and demo some features in WinDbg Preview

Defrag Tools 186 - Jordi and JCAB demo more great features of TTD in WinDbg Preview

CppCon (YouTube) - Jordi, Ken and JamesM presented TTD in WinDbg Preview at CppCon 2017

Trace file basics

Trace file size

The trace file can get big and the user of TTD needs to make sure that there is adequate free space available. If you record a program for even a few minutes, the trace files can quickly grow to be several gigabytes. TTD does not set a maximum size of trace files to allow for complex long running scenarios. Quickly re-creating the issue, will keep the trace file size as small as possible.

Trace and index files

A trace (.RUN) file stores the code execution during recording.

Once the recording is stopped, an index (.IDX) file is created to allow for faster access to the trace information. Index files are also created automatically when WinDbg Preview opens the trace file.

Index files can also be large, typically twice as large as the trace file.

You can recreate the index file from the trace file using the !tt.index command.

```
0:000> !tt.index
Successfully created the index in 10ms.
```

Recording errors and other recording output is written to a WinDbg log file.

All of the output files are stored in a location configured by the user. The default location is in the users document folder. For example, for User1 the TTD files would be stored here:

```
C:\Users\User1\Documents
```

For more information on working the trace files, see Time Travel Debugging - Working with trace files.

Things to look out for

Anti-virus incompatibilities

You may encounter incompatibilities because of how TTD hooks into process to record them. Typically issues arise with anti-virus or other system software that is attempting to track and shadow system memory calls. If you run into issues of with recording, such as an insufficient permission message, try temporarily disabling any anti-virus software.

Other utilities that attempt to block memory access, can also be problematic, for example, the Microsoft Enhanced Mitigation Experience Toolkit. For more information about EMET, see The Enhanced Mitigation Experience Toolkit.

Another example of an environment that conflicts with TTD, would be the electron application framework. In this case the trace may record, but a deadlock or crash of the process being recorded is also possible.

User mode only

TTD currently supports only user mode operation, so tracing a kernel mode process is not possible.

Read-only playback

You can travel back in time, but you can't change history. You can use read memory commands, but you can't use commands that modify or write to memory.

System Protected Processes

Some Windows system protected processes, such as Protected Process Light (PPL) process are protected, so the TTD cannot inject itself into the protected process to allow for the recording of the code execution.

Performance impact of recording

Recording an application or process impacts the performance of the PC. The actual performance overhead varies based upon the amount and type of code being executed during recording. You can expect about a 10x-20x performance hit in typical recording scenarios. Sometimes there will not be a noticeable slowdown but for more resource intensive operations (i.e. File Open dialog) you can see the impact of recording.

Trace file errors

There are some cases where trace file errors can occur. For more information, see Time Travel Debugging - Troubleshooting.

Advanced Features of Time Travel Debugging

Here's some of the most notable TTD advanced features.

Debugger data model support

- **Built in data model support** - TTD includes data model support. Using LINQ queries to analyze application failures can be a powerful tool. You can use the data model window in WinDbg Preview to work with an expandable and browsable version of 'dx' and 'dx -g', letting you create tables using NatVis, JavaScript, and LINQ queries.

For general information about the debugger data model, see WinDbg Preview - Data model. For information about working with the TTD debugger object model, see Time Travel Debugging - Introduction to Time Travel Debugging objects.

Scripting support

- **Scripting Automation** - Scripting support for JavaScript and NatVis allows for the automation of problem investigation. For more information, see Time Travel Debugging - JavaScript Automation.

For general information about working with JavaScript and NatVis, see WinDbg Preview - Scripting.

Managed code TTD support

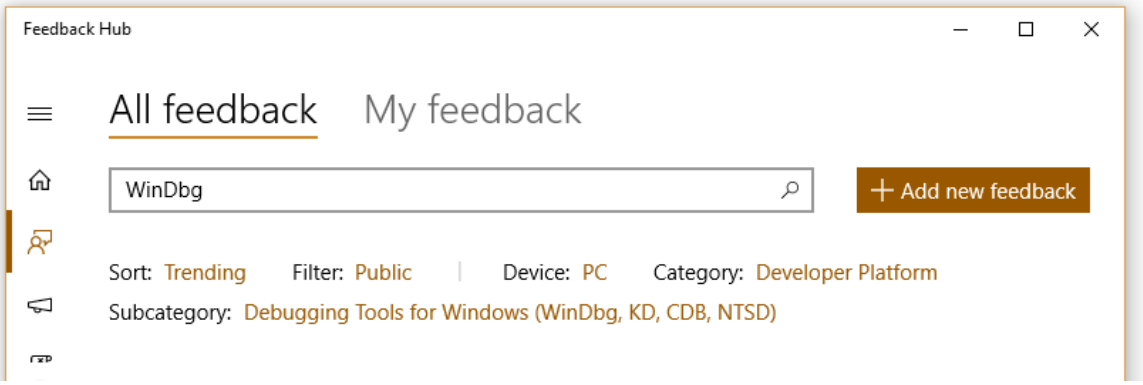
You can use the SOS debugging extension (sos.dll) running in 64 bit mode to debug managed code using TTD

in WinDbg Preview. For more information, see Debugging Managed Code Using the Windows Debugger.

Providing feedback

Your feedback will help guide time travel development priorities going forward.

- If you have feedback such as a feature that you really want to see or a bug that makes something difficult, use the Feedback Hub.



Getting started with TTD

Review these topics to record and replay a trace file as well as for information on working with trace files and troubleshooting.

- Time Travel Debugging - Record a trace
- Time Travel Debugging - Replay a trace
- Time Travel Debugging - Working with trace files
- Time Travel Debugging - Troubleshooting
- Time Travel Debugging - Sample App Walkthrough

These topics describe additional advanced functionality in time travel debugging.

- Time Travel Debugging - Introduction to Time Travel Debugging objects
- Time Travel Debugging - JavaScript Automation

Time Travel Debugging - Record a trace

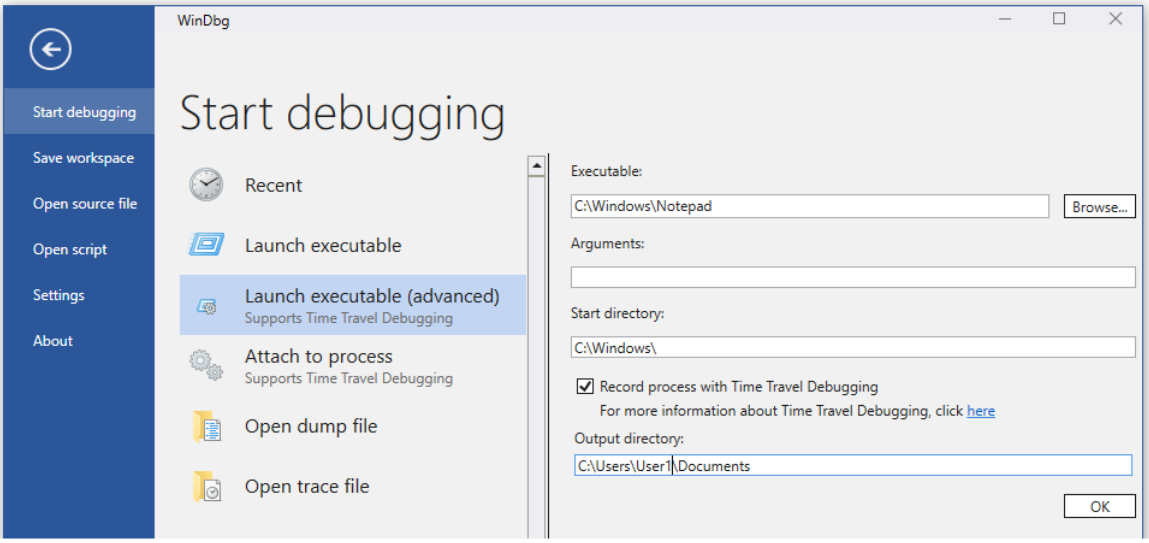
10/25/2019 • 3 minutes to read • Edit Online

This section describes how to record time travel debugging (TTD) traces. There are two ways to record a Trace in WinDbg Preview, *Launch Executable (advanced)* and *Attach to a process*.

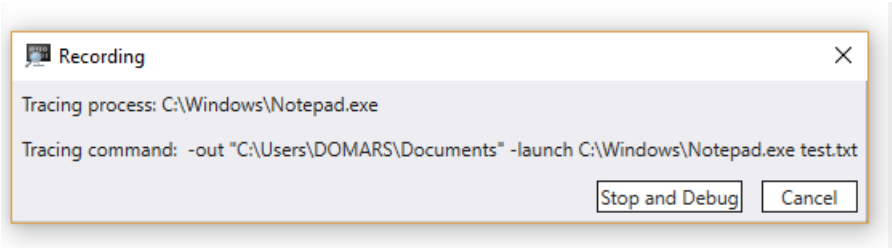
Launch executable (advanced)

To launch an executable and record a TTD trace, follow these steps.

1. In WinDbg Preview, select **File** > **Start debugging** > **Launch executable (advanced)**.
2. Enter the path to the user mode executable that you wish to record or select **Browse** to navigate to the executable. For information about working with the Launch Executable menu in WinDbg Preview, see WinDbg Preview - Start a user-mode session.



3. Check the **Record process with Time Travel Debugging** box to record a trace when the executable is launched.
4. Click **OK** to launch the executable and start recording.
5. The recording dialog appears indicating the trace is being recorded.

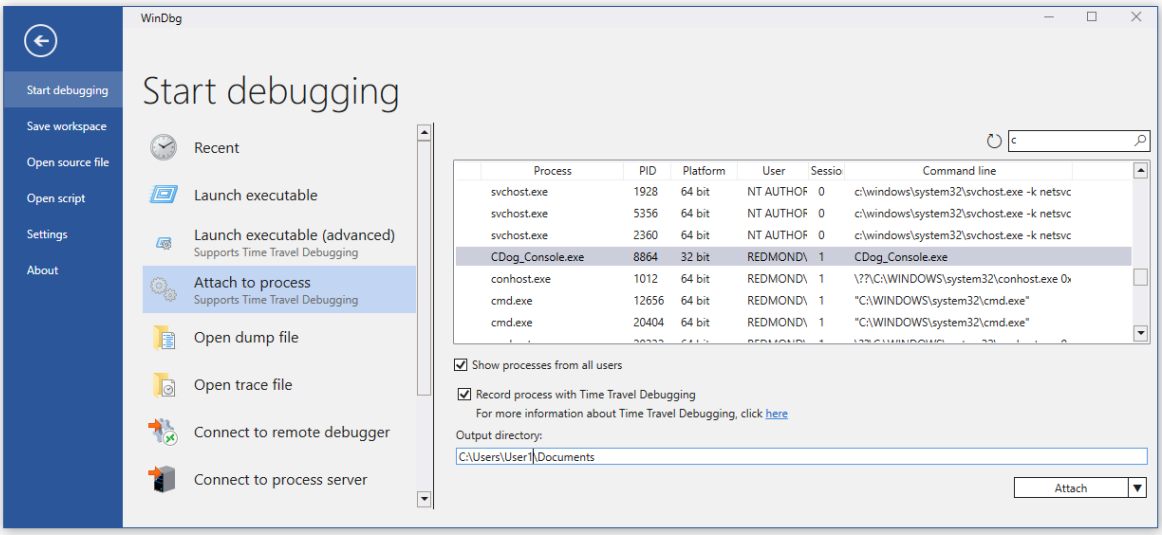


6. See How to record for information on recording.

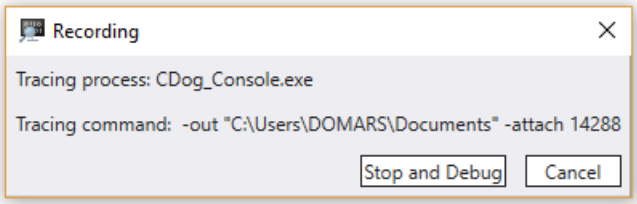
Attach to a process

To attach to a process and record a TTD trace, follow these steps.

1. In WinDbg Preview, select **File** > **Start debugging** > **Attach to process**.
2. Select the user mode process that you wish to trace. For information about working with *Attach to a process* menu in WinDbg Preview, see WinDbg Preview - Start a user-mode session.



3. Check the **Record Process with Time Travel Debugging** box to create a trace when the executable is launched.
4. Click **Attach** to start recording.
5. The recording dialog appears indicating the trace is being recorded.



6. See How to record for information on recording.

How to record

1. The process is being recorded, so this is where you need to cause the issue that you wish to debug. You may open a problematic file or click on a specific button in the app to cause the event of interest to occur.
2. While the recording dialog box is being displayed you can:
   - **Stop and debug** - Choosing this will stop the recording, create the trace file and open the trace file so you can start debugging.
   - **Cancel** - Choosing this will stop the recording and create the trace file. You can open the trace file at a later time.
3. Once your recording is complete, close your app or hit **Stop and debug**.

> **NOTE**
> Both *Stop and debug* and *Cancel* will terminate the associated process.

4. When the application being recorded terminates, the trace file will be closed and written out to disk. This is the case if your program crashes as well.

5. When a trace file is opened, the debugger will automatically index the trace file. Indexing allows for more accurate and faster memory value look ups. This indexing process will take longer for larger trace files.

```
...
00007ffc`61f789d4 c3              ret
0:000> !index
Indexed 1/1 keyframes
Successfully created the index in 96ms.
```

> **NOTE**
> A keyframe is a location in a trace used for indexing. Keyframes are generated automatically. Larger traces will contain more keyframes. When the trace is indexed, the number of keyframes is displayed.

6. At this point you are at the beginning of the trace file and are ready to travel forward and backward in time.

> **TIP**
> Using breakpoints is a common approach to pause code execution at some event of interest. Unique to TTD, you can set a breakpoint and travel back in time until that breakpoint is hit after the trace has been recorded. The ability to examine the process state after an issue has happened, to determine the best location for a breakpoint, enables additional debugging workflows. For an example of using a breakpoint in the past, see Time Travel Debugging - Sample App Walkthrough.

Next Steps

Now that you have a recorded a TTD trace, you can replay the trace back or work with the trace file, for example sharing it with a co-worker. For more information, see these topics.

Time Travel Debugging - Replay a trace

Time Travel Debugging - Working with trace files

Time Travel Debugging - Troubleshooting

Time Travel Debugging - Sample App Walkthrough

See Also

Time Travel Debugging - Overview

Time Travel Debugging - Replay a trace

10/25/2019 • 5 minutes to read • Edit Online

This section describes how to replay time travel traces, navigating forwards and backwards in time.

Command time travel navigation

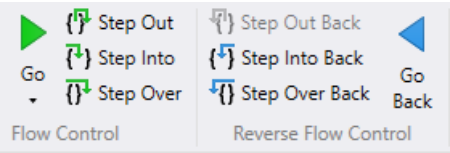
Use a trailing minus sign with the following commands to travel back in time.

COMMAND
p- (Step Back)
t- (Trace Back)
g- (Go Back)

For more information, see Time Travel Debugging - Navigation commands.

Ribbon button time travel navigation

Alternatively, use the ribbon buttons to navigate in the trace.



Example TTD Trace Replay

Use the g- command to execute backwards until either an event or the beginning of the TTD trace is reached. The events that can stop backward execution are the same that would stop forward execution. In this example, the start of the trace is reached.

```
0:000> g-
TTD: Start of trace reached.
(3f78.4274): Break instruction exception - code 80000003 (first/second chance not available)
Time Travel Position: 29:0
ntdll!ZwTestAlert+0x14:
00007ffc`61f789d4 c3              ret
```

Use the p (Step) command to step forward in a TTD trace.

```
0:000> p
Time Travel Position: F:1
eax=0173a5b0 ebx=00fd8000 ecx=7774f821 edx=0f994afc esi=0f99137c edi=00de0000
eip=7774f828 esp=010ff34c ebp=010ff584 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
ntdll!LdrpInitializeProcess+0x1bc5:
7774f828 740b            je      ntdll!LdrpInitializeProcess+0x1bd2 (7774f835) [br=1]
0:000> p
Time Travel Position: F:2
eax=0173a5b0 ebx=00fd8000 ecx=7774f821 edx=0f994afc esi=0f99137c edi=00de0000
eip=7774f835 esp=010ff34c ebp=010ff584 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
ntdll!LdrpInitializeProcess+0x1bd2:
7774f835 83bdd0feffff00  cmp     dword ptr [ebp-130h],0 ss:002b:010ff454=00000000
0:000> p
Time Travel Position: F:3
eax=0173a5b0 ebx=00fd8000 ecx=7774f821 edx=0f994afc esi=0f99137c edi=00de0000
eip=7774f83c esp=010ff34c ebp=010ff584 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
ntdll!LdrpInitializeProcess+0x1bd9:
7774f83c 0f8450e8ffff    je      ntdll!LdrpInitializeProcess+0x42f (7774e092) [br=1]
```

You an also use the t (Trace) command to navigate in the trace.

```
0:000> t
Time Travel Position: F:4
eax=0173a5b0 ebx=00fd8000 ecx=7774f821 edx=0f994afc esi=0f99137c edi=00de0000
eip=7774e092 esp=010ff34c ebp=010ff584 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
ntdll!LdrpInitializeProcess+0x42f:
7774e092 33c0            xor     eax,eax
0:000> t
Time Travel Position: F:5
eax=00000000 ebx=00fd8000 ecx=7774f821 edx=0f994afc esi=0f99137c edi=00de0000
eip=7774e094 esp=010ff34c ebp=010ff584 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
ntdll!LdrpInitializeProcess+0x431:
7774e094 e9f5170000      jmp     ntdll!LdrpInitializeProcess+0x1c2b (7774f88e)
```

Use the p- command to step backwards in a TTD trace.

```
0:000> p-
Time Travel Position: F:4
eax=0173a5b0 ebx=00fd8000 ecx=7774f821 edx=0f994afc esi=0f99137c edi=00de0000
eip=7774e092 esp=010ff34c ebp=010ff584 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
ntdll!LdrpInitializeProcess+0x42f:
7774e092 33c0            xor     eax,eax
0:000> p-
Time Travel Position: F:3
eax=0173a5b0 ebx=00fd8000 ecx=7774f821 edx=0f994afc esi=0f99137c edi=00de0000
eip=7774f83c esp=010ff34c ebp=010ff584 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
ntdll!LdrpInitializeProcess+0x1bd9:
7774f83c 0f8450e8ffff    je      ntdll!LdrpInitializeProcess+0x42f (7774e092) [br=1]
```

You can also use the t- command to navigate backwards in time.

!tt navigation commands

Use the !tt command to navigate forward or backwards in time, by skipping to a given position in the trace.

!tt [position]

Provide a time position in any of the following formats to travel to that point in time.

- If [position] is a decimal number between 0 and 100, it travels to approximately that percent into the trace. For example `!tt 50` travels to halfway through the trace.
- If {position} is #:#, where # are a hexadecimal numbers, it travels to that position. For example, `!tt 1A0:12F` travels to position 1A0:12F in the trace.

For more information, see Time Travel Debugging - !tt (time travel).

!positions

Use `!positions` to display all the active threads, including their position in the trace. For more information, see Time Travel Debugging - !positions (time travel).

```
0:000> !positions
>Thread ID=0x1C74 - Position: F:2
 Thread ID=0x1750 - Position: A5:0
 Thread ID=0x3FFC - Position: 200:0
 Thread ID=0x36B8 - Position: 403:0
 Thread ID=0x3BC4 - Position: 5F2:0
 Thread ID=0x392C - Position: B45:0
 Thread ID=0x32B4 - Position: C87:0
 Thread ID=0x337C - Position: DF1:0
```

This example shows that there are eight threads at the current position. The current thread is 3604, marked with '>'.

TIP

Another way to display the current list of threads and their positions, is to use the data model dx command:

```
dx -g @$curprocess.Threads.Select(t => new { IsCurrent = t.Id == @$curthread.Id, ThreadId = t.Id, Position = t.TTD.Position })
```

Use the user mode ~ (Thread Status) command shows the same eight threads, and marks the current thread with '.':

```
0:000> ~
.  0  Id: 954.1c74 Suspend: 4096 Teb: 00fdb000 Unfrozen
   1  Id: 954.1750 Suspend: 4096 Teb: 00fea000 Unfrozen
   2  Id: 954.3ffc Suspend: 4096 Teb: 00fde000 Unfrozen
   3  Id: 954.36b8 Suspend: 4096 Teb: 00fe1000 Unfrozen
   4  Id: 954.3bc4 Suspend: 4096 Teb: 00fe4000 Unfrozen
   5  Id: 954.392c Suspend: 4096 Teb: 00fed000 Unfrozen
   6  Id: 954.32b4 Suspend: 4096 Teb: 00ff0000 Unfrozen
   7  Id: 954.337c Suspend: 4096 Teb: 00ff3000 Unfrozen
```

Click on the link next to the third thread (3FFC) in the !positions output, to time travel to that position in the trace, 200:0.

```
0:002> !ttdext.tt 200:0
Setting position: 200:0
(954.3ffc): Break instruction exception - code 80000003 (first/second chance not available)
Time Travel Position: 200:0
eax=00000000 ebx=012da718 ecx=7775396c edx=00000000 esi=012e1848 edi=012e1a08
eip=7775396c esp=014cf9f8 ebp=014cfbfc iopl=0         nv up ei ng nz ac po cy
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000293
ntdll!NtWaitForWorkViaWorkerFactory+0xc:
7775396c c21400          ret     14h
```

Use the ~ (Thread Status) command to confirm that we are now positioned at the third thread, 3ffc.

```
0:002> ~
   0  Id: 954.1c74 Suspend: 4096 Teb: 00fdb000 Unfrozen
   1  Id: 954.1750 Suspend: 4096 Teb: 00fea000 Unfrozen
.  2  Id: 954.3ffc Suspend: 4096 Teb: 00fde000 Unfrozen
   3  Id: 954.36b8 Suspend: 4096 Teb: 00fe1000 Unfrozen
   4  Id: 954.3bc4 Suspend: 4096 Teb: 00fe4000 Unfrozen
   5  Id: 954.392c Suspend: 4096 Teb: 00fed000 Unfrozen
   6  Id: 954.32b4 Suspend: 4096 Teb: 00ff0000 Unfrozen
   7  Id: 954.337c Suspend: 4096 Teb: 00ff3000 Unfrozen
```

NOTE

The *~s#*, where # is a thread number, also switches to the given thread, but it doesn't change the current position in the trace. When *!tt* is used to time travel to another thread's position, any values you (and the debugger) read from memory will be looked up at that position. When switching threads with *~s#*, the debugger doesn't change the current position internally, which is used for all memory queries. This works this way primarily so that *~s#* doesn't have to reset the debugger's inner loop.

Time travel debugging extension commands

For information on the `!tt`, `!positions` and the `!index` commands see Time Travel Debugging - Extension Commands.

See Also

Time Travel Debugging - Overview

Time Travel Debugging - Record a trace

Time Travel Debugging - Working with trace files

Time Travel Debugging - Sample App Walkthrough

Time Travel Debugging - Working with Trace Files

10/25/2019 • 7 minutes to read • Edit Online

This section describes how to work with files created and consumed by time travel debugging.

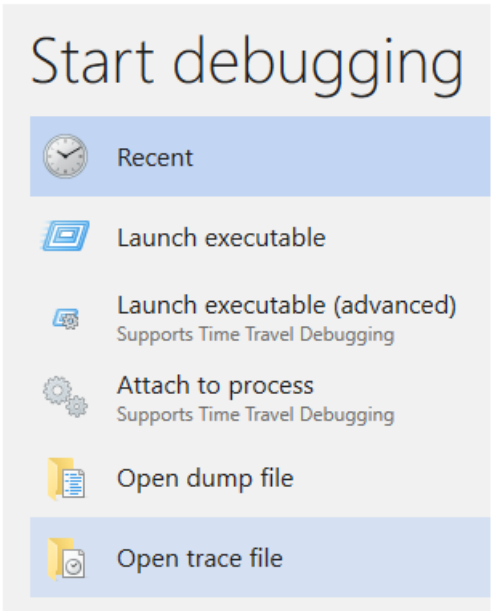
Trace File Overview

Time Travel Debugging uses the following files to debug code execution.

- The trace file contains the code execution recording and has a .RUN extension.
- The index file enables quick access to information in the Trace file and has an .IDX extension.
- Recording errors and other recording output is written to the debugger log file.

Trace .RUN files

Trace .RUN files can be opened after they are recorded using **File** > **Start debugging** > **Open trace file**.

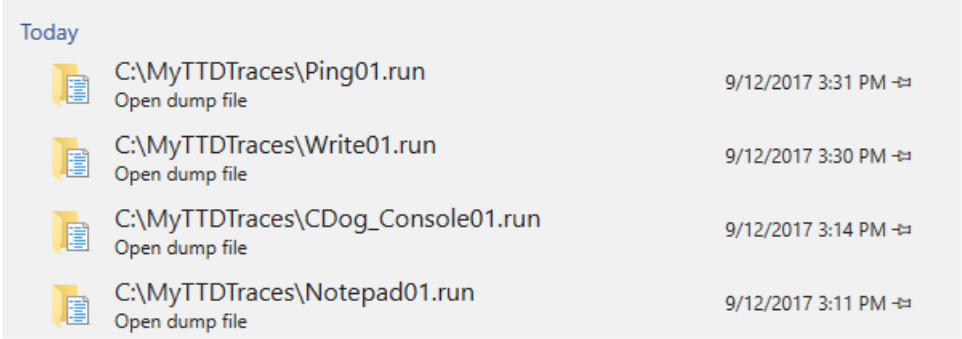


All of the trace output files are stored in the users document folder by default. For example, for User1 the TTD files would be stored here:

```
C:\Users\User1\Documents
```

You can change the location of the trace files when you start to record. For more information, see Time Travel Debugging - Recording.

The most recently used list of files allows you to quickly access previously used target configuration files. Any recently used trace files or dump files are listed as well.



Index .IDX files

An index .IDX file is created for the associated trace .RUN file automatically when opening the trace file in WinDbg Preview. You can manually create the index file by using the !index command. An index allows for faster access to the trace information.

IDX files can also be large, typically twice the size of the .RUN file.

Recreating the .IDX file

You can recreate the .IDX file from the .RUN file, using the `!index` command. For more information, see Time Travel Debugging - !index (time travel).

```
0:0:001> !index
Indexed 3/3 keyframes
Successfully created the index in 49ms.
```

Sharing TTD Trace .RUN files

TTD is local only and does not work remotely connected to another machine.

TTD trace files can be shared with others by copying the .RUN file. This can be handy for having a coworker help you figure out the problem. They don't need to install the crashing app or do any other related setup to attempt to reproduce the issue. They can just load the trace file and debug the app as if it was installed on their PC.

The machine where you replay the TTD trace must support all instructions that were used on the record machine - for example AVX instructions.

You can rename the file to include any additional information, such as the date or a bug number.

The .IDX file does not need to be copied as it can be re-created using the !index command as described above.

> **TIP**
> When collaborating with others, pass on any relevant trace positions related to the problem at hand. The collaborator can use the `!tt x:y` command to move to that exact point in time in the execution of the code. Time position ranges can be included in bug descriptions to track where the possible issue may be occurring.

Error - Log File

Recording errors and other recording output is written to the debugger log file. To view the log file, select **View** > **Logs**.

This example shows the error log text when attempting to launch and record an executable named Foo.exe that is not in the C:\Windows directory.

```
2017-09-21:17:18:10:320 : Information : DbgXUI.dll : TTD: Output:
Microsoft (R) TTD 1.01.02
Release: 10.0.16366.1000
Copyright (C) Microsoft Corporation. All rights reserved.
Launching C:\Windows\Foo.exe
2017-09-21:17:18:10:320 : Error : DbgXUI.dll : TTD: Errors:
Error: Trace of C:\Windows\Foo.exe PID:0 did not complete successfully: status:27
Error: Could not open 'Foo.exe'; file not found.
Error: Corrupted trace dumped to C:\Users\User1\Documents\Foo01.run.err.
```

Trace file size

TTD trace files can get really big and it is important to ensure that you have adequate free disk space available. If you record an app or process even just for a few minutes, the trace file may grow to be several gigabytes in size. The size of the trace file is dependent on a number of factors described below.

TTD does not set a maximum size of trace files to allow for complex long running scenarios. Quickly re-creating the issue, will keep the trace file size as small as possible.

Trace file size factors

It is not possible to provide and exact estimate of the the trace file size but there are a few rules of thumb to help you understand TTD file sizes.

The following factors can affect the size of the trace file:

- The number of code instructions executed across all threads when recording the running app or process
- The length of time during which the app or process was recorded (only as this affects the number of code instructions recorded)
- The size of the memory data used by the app or process

The number of instructions executed and recorded is the biggest factor that affects the trace file size. A trace typically requires between 1 bit and 1 byte per instruction executed. A trace is likely to be towards the lower end of that range when the recorded program executes a smaller number of distinct functions and operates on a smaller set of data. A trace is likely to be towards the higher end of that range when the recorded program executes a larger number of distinct functions or operates on a larger set of data.

Trace file size rule of thumb

The trace file grows roughly 5MB to 50MB per second when recording an active app or process, depending on the trace file size factors identified above.

The trace file will not grow when the app or process being recorded is idle (e.g., when waiting for input).

Currently there is no maximum file size limit for trace files. WinDbg Preview can replay trace files that are well into the hundreds of gigabytes in size.

Index file size

The index file is automatically created by WinDbg Preview when you open a trace for the first time. It contains information that helps the debugger to replay the trace and query information about memory more efficiently. Its size generally ranges from 1 to 2 times the size of the trace file. The factors that affect its size are similar to those that affect the size of the trace file.

First, the size of the index file scales relative to the length of the trace. A trace that contains a larger number of recorded instructions will generally have a larger index.

Second, the size of the index scales relative to the breadth of the memory accesses. If the program that was recorded frequently accessed a large number of distinct memory locations, the index will generally be larger than if the program that was recorded accessed fewer distinct memory locations or if accesses to memory locations was less frequent.

Since these factors are similar to the factors that affect the size of the trace file, the size of the index file generally scales relative to the size of the index file (thus our estimate that it's typically between 1x and 2x the size of the trace file).

What if I run out of disk space?

Both the TTD trace and index files are written to disk. Currently there is not a maximum file size limitation for either the trace or index file. The trace file grows in size until you stop recording or exceed the amount of available disk space.

During recording: TTD will write out last page to the trace file and then effectively wait until it can write again. WinDbg continues to show the Recording dialog but does not show an error/warning message when running out of disk space during recording.

Running out of disk space during recording results in a trace file with an incomplete record of the code execution. The incomplete trace file can be opened in WinDbg Preview but it may not include the actual problem if the error occurs after running out of disk space when writing the trace file.

Workaround: Open File Explorer and check if the disk (i.e. C: drive) Free space is near zero. Alternately watch the trace (.RUN) file in File Explorer (default in Documents folder) and if not regularly growing in size then recording may be waiting. Click the Stop and Debug button in WinDbg, free up space or save to another disk, and start recording again.

During indexing: The debugger may produce an invalid index file, resulting in unpredictable behavior in the debugger, or the debugger engine host may crash.

Workaround: Close the debugger and delete any index file (.idx) that may exist for your trace. Either free up sufficient disk space, or move the trace file to a different disk with sufficient free space. Open the trace again in the debugger and run !index to create a new, correct index. Indexing does not modify the original trace file (.run), so no data will have been lost.

See Also

Time Travel Debugging - Overview

Time Travel Debugging - Troubleshooting

10/25/2019 • 4 minutes to read • Edit Online

This section describes how to troubleshoot time travel traces.

Issues attempting to record a process

I get an error message that says "WinDbg must be run elevated to support Time Travel Debugging"

As the message indicates, running the debugger elevated is a requirement. In order to run the debugger elevated, right-click on the **WinDbg Preview** icon in the start menu and then select **More** > **Run as Administrator**.

I installed WinDbg Preview with an account that does not have administrator privileges and I get an error message that says "WinDbg must be run elevated to support Time Travel Debugging"

Re-install WinDbg Preview using an account that has administrator privileges and use that account when recording in the debugger.

I can't launch and record a UWP application

This is not supported at this time, but you may attach to and record an already-running UWP application.

I can't record a <insert name of unusual process type - running in another session, security context, credentials...> process

At this time, TTD only record regular processes that you can be normally launched from a command console or by clicking on an executable or shortcut in Windows Explorer.

I cannot successfully record my application on my computer

If recording of your application fails, verify that you can record a simple Windows process. For example, "ping.exe" or "cmd.exe" are simple processes that can normally be recorded.

I cannot successfully record anything at all on my computer

TTD recording is an invasive technology, which can interfere with other invasive technologies like application virtualization frameworks, information management products, security software or antivirus products.

See "Things to look out for" in Time Travel Debugging - Overview for information on known TTD incompatibilities.

I'm tracing an application and running AppVerifer at the same time, and the performance when replaying the trace is slow.

Because of the way AppVerifer uses memory to check the application, the experience later when replaying the trace can be noticeably worse than without AppVerifier. To improve performance, disable AppVerifier when recording the app. If this is not possible, you may need to close the callstack window in WinDbg in order to improve performance.

Issues with .IDX index files

Debugging a trace file without an index file, or with a corrupted or incomplete index file, is possible, but is not recommended. The index file is needed to ensure that memory values read from the debugged process are most accurate, and to increase the efficiency of all other debugging operations.

Use the `!index -status` command to examine the state of the .IDX index file associated with the .RUN trace file.

If it you may try recreating the index file by running `!index -force`.

Recreating the .IDX index file

If you suspect and issue with the index file, or `!index -status` says anything other than "Index file loaded", recreate it. To do this you may run `!index -force`. If that fails:

1. Close the debugger.
2. Delete the existing IDX file, it will have the same name as the .RUN trace file and be located in the same directory that the .RUN file is.
3. Open the trace .RUN file in WinDbg Preview. This will run the `!index` command to re-create the index.
4. Use the `!index -status` command to confirm that the trace index is functional.

Ensure that there's enough space for the index file in the same location where the trace file resides. Depending on the contents of the recording, the index file may be significantly larger than the trace file, typically on the order of twice as large.

Issues with Trace .RUN Files

When there are issues with the trace .RUN file, you may receive error messages such as these.

```
Replay and log are out of sync at fallback data. Packet type is incorrect "Packet Type"
Replay and log are out of sync at opaque data. Log had already reached the end
Replay exit thread event does not match up with logged event
Logged debug write values are out of sync with replay
```

In most cases all of the failure messages indicate that the .RUN trace file is not usable and must be re-recorded.

Re-recording the user mode app

If there is a specific issue with recording a user mode app, you may want to try recording a different app on the same PC, or try the same app on a different PC. You may want to try and record a different use of the app to see if there is a specific issue with recording certain parts of the app.

When debugging or creating the index, I see messages about “Derailment events”.

It is possible that you may see messages like this one:

```
Derailment event MissingDataDerailment(7) on UTID 2, position 2A550B:108 with PC 0x7FFE5EEB4448 Request
address: 0x600020, size: 32
```

TTD works by running an emulator inside of the debugger, which executes the instructions of the debugged process in order to replicate the state of that process at every position in the recording. Derailments happen when this emulator observes some sort of discrepancy between the resulting state and information found in the trace file. The error quoted above, for instance, refers to an instruction found on location 0x7FFE5EEB4448, at position 2A550B:108 in the trace, which attempted to read some memory around location 0x600020, which doesn’t exist in the recording.

Derailments are often caused by some error in the recorder, or sometimes in the emulator, at some recorded instruction further back in the trace.

In most cases this failure message indicates that the .RUN trace file will have a gap in the thread that derailed, starting at the point that it derailed, for some indeterminate number of instructions. If the event of interest you are trying to debug didn’t happen during that gap, the trace may be usable. If the event of interest occurred during that gap, the trace will need to be re-recorded.

See Also

Time Travel Debugging - Overview

Time travel navigation commands

10/25/2019 • 2 minutes to read • Edit Online

This section describes the time travel navigation commands.

p- (Step Back)

The *p-* command executes the previous single instruction or source line. When subroutine calls or interrupts occur, they are treated as a single step. You can invoke this command using the **Step Over Back** button on the **Home** ribbon in WinDbg Preview.

t- (Trace Back)

The *t-* command executes the previous single instruction or source line. When subroutine calls or interrupts occur, each of their steps is also traced. You can invoke this command using the **Step Into Back** button on the **Home** ribbon in WinDbg Preview.

g- (Go Back)

The *g-* command starts executing the current process in reverse. Execution will halt at the end of the program, when BreakAddress is hit, or when another event causes the debugger to stop. You can invoke this command using the **Go Back** button on the **Home** ribbon in WinDbg Preview.

Additional Information

The time travel navigation commands only work with time travel traces. For more information about time travel, see Time Travel Debugging - Overview.

See Also

Time Travel Debugging - Overview

Time Travel Debugging - Replay a trace

Time travel debugging extension commands

10/25/2019 • 2 minutes to read • Edit Online

This section introduces the time travel debugger extension commands.

In this section

TOPIC	DESCRIPTION
!tt (time travel)	The **!tt (time travel)** debugger extension that allows you to navigate forward and backwards in time.
!positions	The **!positions** extension displays all the active threads, including their current positions in the time travel trace.
!index	The **!index** extension indexes time travel traces or displays index status information.

Additional Information

This extension only works with time travel traces. For more information about time travel, see Time Travel Debugging - Overview.

DLL

The time travel debugger extension commands are implemented in ttdext.dll. The time travel command dll is loaded automatically in WinDbg Preview. You don't need to use the load command to manually load the ttdext.dll.

See Also

Time Travel Debugging - Overview

!tt (time travel)

10/25/2019 • 2 minutes to read • Edit Online

The !tt (time travel) debugger extension that allows you to navigate forward and backwards in time.

!tt navigation commands

Use the !tt extension to navigate forward or backwards in time, by traveling to a given position in the trace.

```
!tt [position]
```

Parameters

position

Provide a time position in any of the following formats to travel to that point in time.

- If {position} is a decimal number between 0 and 100, it travels to approximately that percent into the trace. For example:
  - !ttdext.tt 0 - Time travel to the beginning of the trace
  - !ttdext.tt 50 - Time travel to halfway through the trace
  - !ttdext.tt 100 - Time travel to the end of the trace
- If {position} is #:#, where # are a hexadecimal numbers, it travels to that position. If the number after : is omitted, it defaults to zero.
  - !ttdext.tt 1A0: - Time travel to position 1A0:0
  - !ttdext.tt 1A0:0 - Time travel to position 1A0:0
  - !ttdext.tt 1A0:12F - Time travel to position 1A0:12F

> **NOTE**
>
> Traces use a two-part instruction position that references a specific position reference in the trace, for example 12:0. or 15:7. The two elements are hexadecimal numbers defined as described here.
>
> xx:yy
>
> xx- the first element is the sequencing number, which corresponds to a sequencing event.
>
> yy - the second element is a step count, which corresponds roughly to the instruction count since the sequencing event.

DLL

ttdext.dll

Additional Information

This extension only works with time travel traces. For more information about time travel, see Time Travel

Debugging - Overview.

See Also

Time Travel Debugging - Overview

!index

10/25/2019 • 2 minutes to read • Edit Online

The **!index** extension indexes time travel traces or displays index status information.

```
!index [-status] [-force]
```

Use `!index` to run an indexing pass over the current trace.

```
0:000> !index
Indexed 10/14 keyframes
Indexed 14/14 keyframes
Successfully created the index in 535ms.
```

If the current trace is already indexed, the !index command does nothing.

```
0:000> !index
Successfully created the index in 0ms.
```

Parameters

-status

Use `!index -status` to report the status of the trace index.

```
0:000> !tt.index -status
Index file loaded.
```

-force

Use `!index -force` to reindex the trace even if an unloadable index file exists on disk.

```
0:000> !tt.index -force
Successfully created the index in 152ms.
```

DLL

ttdext.dll

Additional Information

This extension only works with time travel traces. For more information about time travel, see Time Travel Debugging - Overview.

See Also

Time Travel Debugging - Extension Commands

!positions

10/25/2019 • 2 minutes to read • Edit Online

The **!positions** extension displays all the active threads, including their current positions in the time travel trace.

```
!positions
```

Parameters

None

Example

This output shows four threads. Thread 3824 is the current thread indicated by **>** in the left column.

```
0:000> !positions
>Thread ID=0x3824 - Position: F:0
 Thread ID=0x2684 - Position: A5:0
 Thread ID=0x2478 - Position: 200:0
 Thread ID=0x2DC4 - Position: 401:0
```

DLL

ttdext.dll

Additional Information

This extension only works with time travel traces. For more information about time travel, see Time Travel Debugging - Overview.

See Also

Time Travel Debugging - Extension Commands

Time Travel Debugging - Sample App Walkthrough

10/25/2019 • 19 minutes to read • Edit Online

This lab introduces Time Travel Debugging (TTD), using a small sample program with a code flaw. TTD is used to debug, identify and root cause the issue. Although the issue in this small program is easy to find, the general procedure can be used on more complex code. This general procedure can be summarized as follows.

1. Capture a time travel trace of the failed program.
2. Use the dx (Display Debugger Object Model Expression) command to find the exception event stored in the recording.
3. Use the !tt (time travel) command to travel to the position of the exception event in the trace.
4. From that point in the trace single step backwards until the faulting code in question comes into scope.
5. With the faulting code in scope, look at the local values and develop a hypothesis of a variable that may contain an incorrect value.
6. Determine the memory address of the variable with the incorrect value.
7. Set a memory access (ba) breakpoint on the address of the suspect variable using the ba (Break on Access) command.
8. Use g- to run back to last point of memory access of the suspect variable.
9. See if that location, or a few instructions before, is the point of the code flaw. If so, you are done. If the incorrect value came from some other variable, set another break on access breakpoint on the second variable.
10. Use g- to run back to the last point of memory access on the second suspect variable. See if that location or a few instructions before contains the code flaw. If so, you are done.
11. Repeat this process walking back until the code that set the incorrect value that caused the error is located.

Although the general techniques described in this procedure apply to a broad set of code issues, there are unique code issues that will require a unique approach. The techniques illustrated in the walkthrough should serve to expand your debugging tool set and will illustrate some of what is possible with a TTD trace.

Lab objectives

After completing this lab, you will be able to use the general procedure with a time travel trace to locate issues in code.

Lab setup

You will need the following hardware to be able to complete the lab.

- A laptop or desktop computer (host) running Windows 10

You will need the following software to be able to complete the lab.

- The WinDbg Preview. For information on installing WinDbg Preview, see WinDbg Preview - Installation
- Visual Studio to build the sample C++ code.

The lab has the following three sections.

- Section 1: Build the sample code
- Section 2: Record a trace of the "DisplayGreeting" sample
- Section 3: Analyze the trace file recording to identify the code issue

Section 1: Build the sample code

In Section 1, you will build the sample code using Visual Studio.

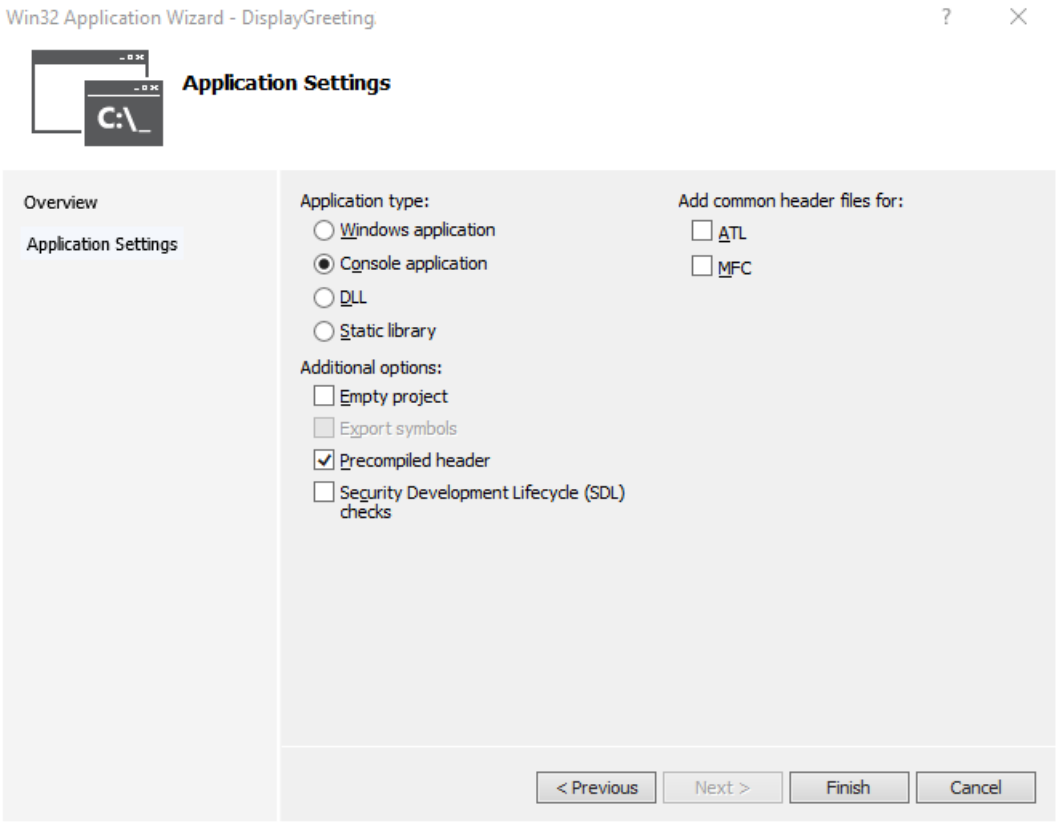
Create the sample app in Visual Studio

1. In Microsoft Visual Studio, click **File** > **New** > **Project/Solution...** and click on the Visual **C++** templates.

   Select the Win32 Console Application.

   Provide a project name of *DisplayGreeting* and click on **OK**.

2. Uncheck the Security Development Lifecycle (SDL) checks.



3. Click on **Finish**.

4. Paste in the following text to the DisplayGreeting.cpp pane in Visual Studio.

```
// DisplayGreeting.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include <array>
#include <stdio.h>
#include <string.h>

void GetCppConGreeting(wchar_t* buffer, size_t size)
{
    wchar_t const* const message = L"HELLO FROM THE WINDBG TEAM. GOOD LUCK IN ALL OF YOUR TIME TRAVEL
DEBUGGING!";

    wcscpy_s(buffer, size, message);
}

int main()
{
     std::array <wchar_t, 50> greeting{};
     GetCppConGreeting(greeting.data(), sizeof(greeting));

     wprintf(L"%ls\n", greeting.data());

     return 0;
}
```

5. In Visual Studio, click **Project** > **DisplayGreeting properties**. Then click on **C/C++** and **Code Generation**.

   Set the following properties.

SETTING	VALUE
Security Check	Disable Security Check (/GS-)
Basic Runtime Checks	Default

> **NOTE**
> Although these setting are not recommended, it is possible to imagine a scenario where someone would advise using these settings to expedite coding or to facilitate certain testing environments.

5. In Visual Studio, click **Build** > **Build Solution**.

   If all goes well, the build windows should display a message indicating that the build succeeded.

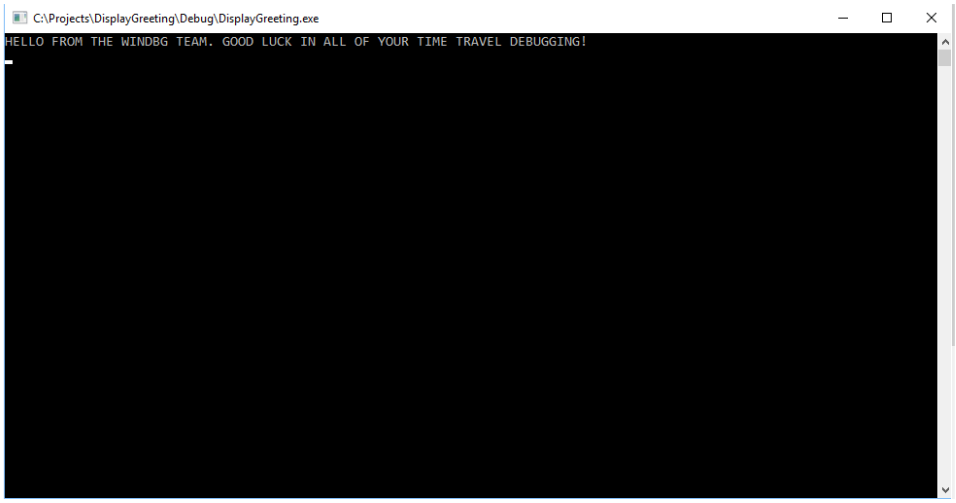
6. **Locate the built sample app files**

   In the Solution Explorer, right click on the *DisplayGreeting* project and select **Open Folder in File explorer**.

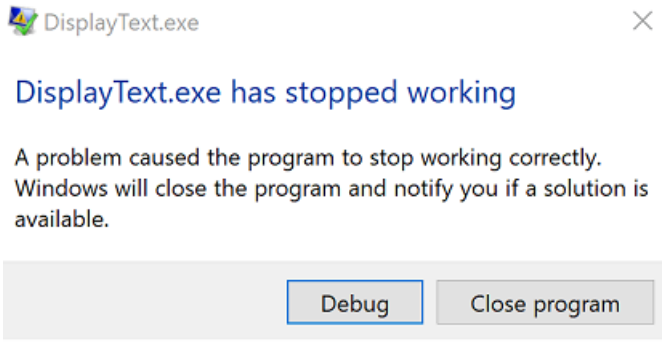
   Navigate to the Debug folder that contains the complied exe and symbol pdb file for the sample. For example, you would navigate to *C:\Projects\DisplayGreeting\Debug*, if that's the folder that your projects are stored in.

7. **Run the sample app with the code flaw**

   Double click on the exe file to run the sample app.



If this dialog box appears, select **Close program**



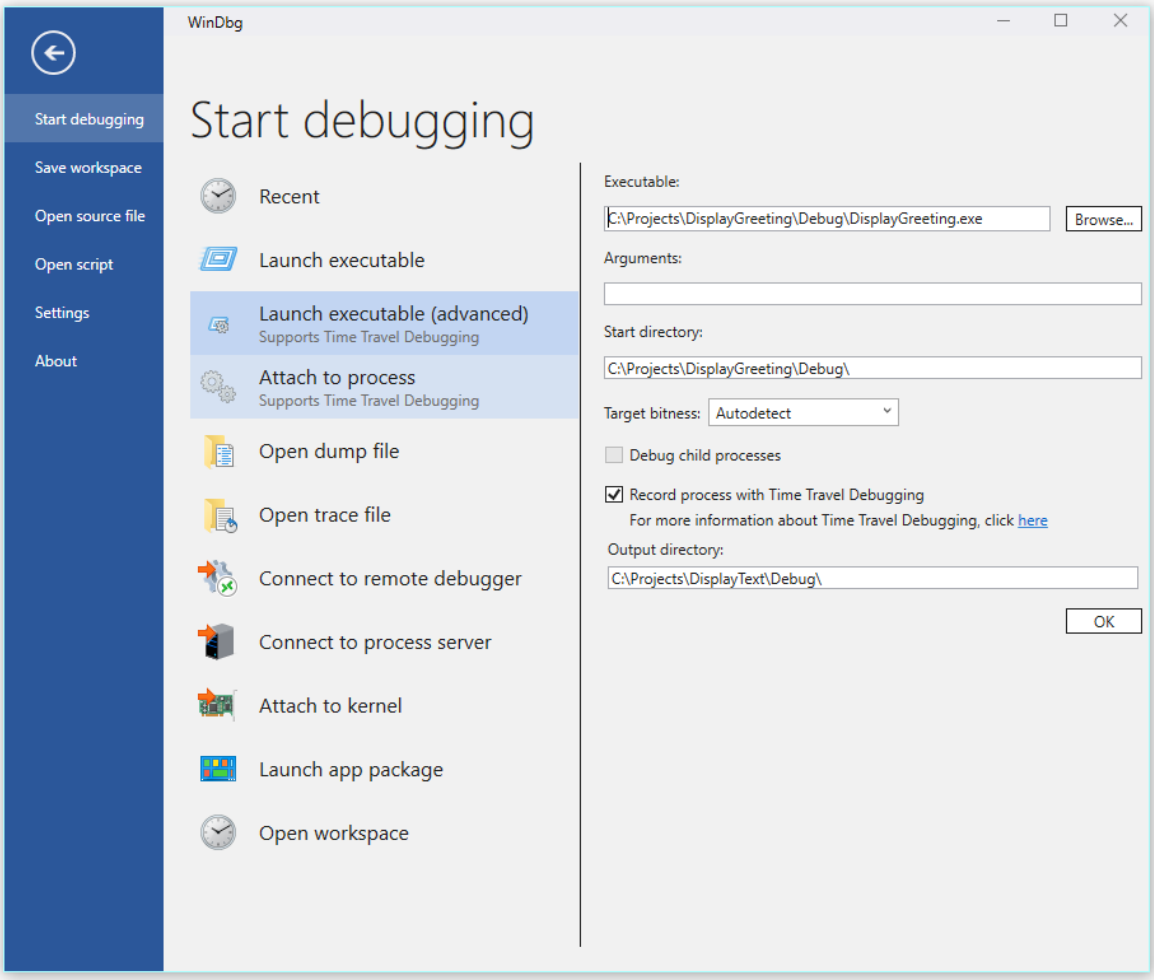
In the next section of the walkthrough, we will record the execution of the sample app to see if we can determine why this exception is occurring.

Section 2: Record a trace of the "DisplayGreeting" sample

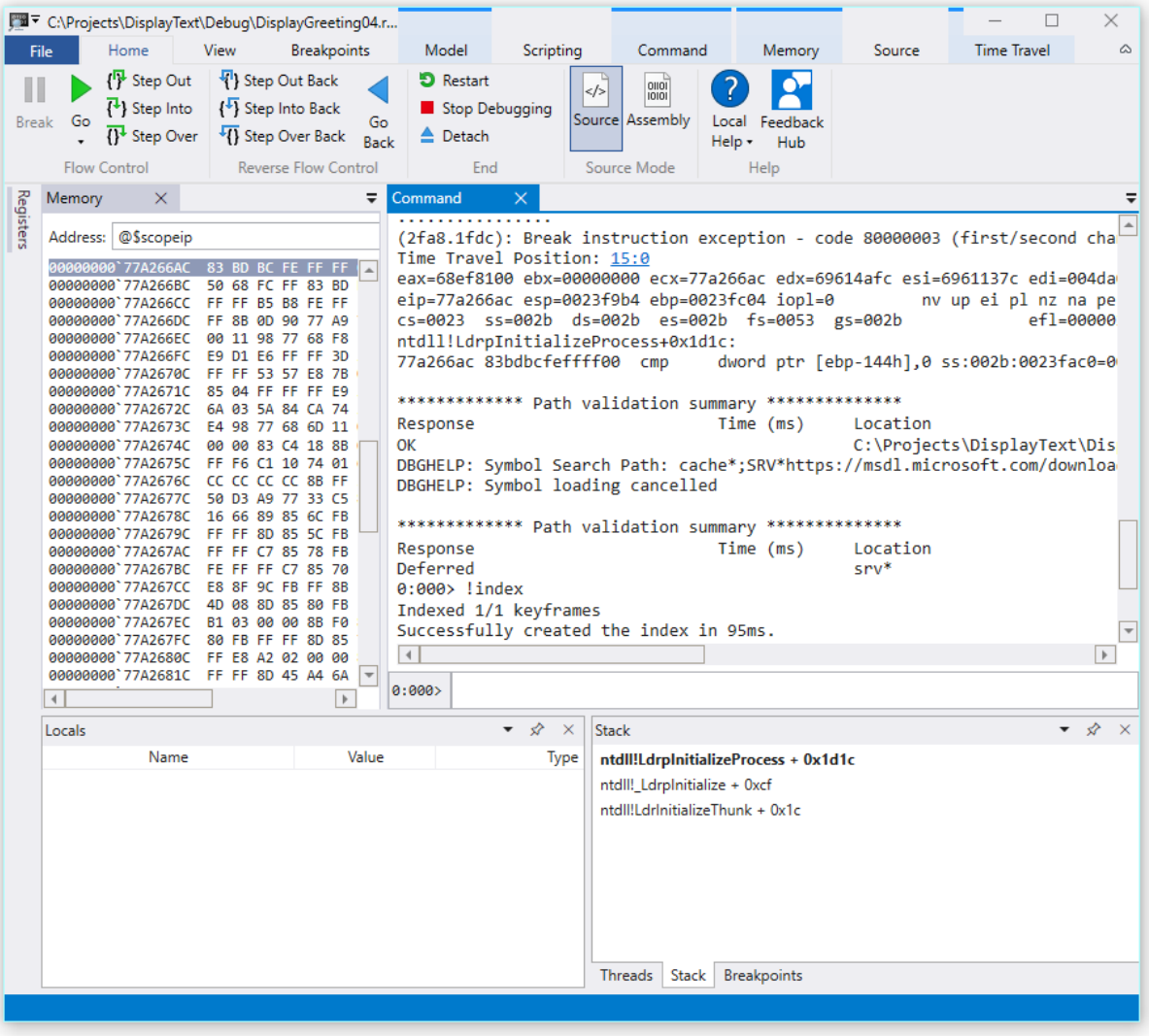
In Section 2, you will record a trace of the misbehaving sample "DisplayGreeting" app

To launch the sample app and record a TTD trace, follow these steps. For general information about recording TTD traces, see Time Travel Debugging - Record a trace

1. Run WinDbg Preview as an Administrator, so as to be able to record time travel traces.
2. In WinDbg Preview, select **File** > **Start debugging** > **Launch executable (advanced)**.
3. Enter the path to the user mode executable that you wish to record or select **Browse** to navigate to the executable. For information about working with the launch executable menu in WinDbg Preview, see WinDbg Preview - Start a user-mode session.



4. Check the **Record process with Time Travel Debugging** box to record a trace when the executable is launched.
5. Click **OK** to launch the executable and start recording.
6. The recording dialog appears indicating the trace is being recorded. Shortly after that, the application crashes.
7. Click on **Retry**, to allow the code to try and run.
8. The program crashes and the trace file will be closed and written out to disk.



9. The debugger will automatically open the trace file and index it. Indexing is a process that enables efficient debugging of the trace file. This indexing process will take longer for larger trace files.

```
0:000> !index
Indexed 1/1 keyframes
Successfully created the index in 95ms.
```

> **NOTE**
>
> A keyframe is a location in a trace used for indexing. Keyframes are generated automatically. Larger traces will contain more keyframes.

10. At this point you are at the beginning of the trace file and are ready to travel forward and backward in time.

    Now that you have a recorded a TTD trace, you can replay the trace back or work with the trace file, for example sharing it with a co-worker. For more information about working with trace files, see Time Travel Debugging - Working with Trace Files

In the next section of this lab we will analyze the trace file to locate the issue with our code.

Section 3: Analyze the trace file recording to identify the code issue

In Section 3, you will analyze the trace file recording to identify the code issue.

Configure the WinDbg Environment

1. Add your local symbol location to the symbol path and reload the symbols, by typing the following commands.

```
.sympath+ C:\Projects\DisplayGreeting\Debug
.reload
```

2. Add your local code location to the source path by typing the following command.

```
.srcpath C:\Projects\DisplayGreeting\DisplayGreeting
```

3. To be able to view the state of the stack and local variables, on the WinDbg Preview ribbon, select **View** and **Locals** and **View** and **Stack**. Organize the windows to allow you to view them, the source code and the command windows at the same time.

4. On the WinDbg Preview ribbon, select **Source** and **Open Source File**. Locate the DisplayGreeting.cpp file and open it.

Examine the exception

1. When the trace file was loaded it displays information that an exception occurred.

```
2fa8.1fdc): Break instruction exception - code 80000003 (first/second chance not available)
Time Travel Position: 15:0
eax=68ef8100 ebx=00000000 ecx=77a266ac edx=69614afc esi=6961137c edi=004da000
eip=77a266ac esp=0023f9b4 ebp=0023fc04 iopl=0         nv up ei pl nz na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000206
ntdll!LdrpInitializeProcess+0x1d1c:
77a266ac 83bdbcfeffff00  cmp     dword ptr [ebp-144h],0 ss:002b:0023fac0=00000000
```

2. Use the dx command to list all of the events in the recording. The exception event is listed in the events.

```
0:000> dx -r1 @$curprocess.TTD.Events
...
[0x2c]           : Module Loaded at position: 9967:0
[0x2d]           : Exception at 9BDC:0
[0x2e]           : Thread terminated at 9C43:0
...
```

> **NOTE**
> In this walkthrough three periods are used to indicate that extraneous output was removed.

3. Click on the Exception event to display information about that TTD event.

```
0:000> dx -r1 @$curprocess.TTD.Events[17]
@$curprocess.TTD.Events[17]                : Exception at 68:0
    Type             : Exception
    Position         : 68:0 [Time Travel]
    Exception        : Exception of type Hardware at PC: 0X540020
```

4. Click on the Exception field to further drill down on the exception data.

```
0:000> dx -r1 @$curprocess.TTD.Events[17].Exception
@$curprocess.TTD.Events[17].Exception                : Exception of type Hardware at PC: 0X540020
    Position         : 68:0 [Time Travel]
    Type             : Hardware
    ProgramCounter   : 0x540020
    Code             : 0xc0000005
    Flags            : 0x0
    RecordAddress    : 0x0
```

The exception data indicates that this is a Hardware fault thrown by the CPU. It also provides the exception code of 0xc0000005 that indicates that this is an access violation. This typically indicates that we were attempting to write to memory that we don't have access to.

5. Click on the [Time Travel] link in the exception event to move to that position in the trace.

```
0:000> dx @$curprocess.TTD.Events[17].Exception.Position.SeekTo()
Setting position: 68:0

@$curprocess.TTD.Events[17].Exception.Position.SeekTo()
(16c8.1f28): Break instruction exception - code 80000003 (first/second chance not available)
Time Travel Position: 68:0
eax=00000000 ebx=00cf8000 ecx=99da9203 edx=69cf1a6c esi=00191046 edi=00191046
eip=00540020 esp=00effe4c ebp=00520055 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
00540020 ??
```

Of note in this output is that the stack and base pointer are pointing to two very different addresses.

```
esp=00effe4c ebp=00520055
```

This could indicate that stack corruption - possibly a function returned and then corrupted the stack. To validate this, we need to travel back to before the CPU state was corrupted and see if we can determine when the stack corruption occurred.

Examine the local variables and set a code breakpoint

At the point of failure in trace it is common to end up a fews steps after the true cause in error handling code. With time travel we can go back an instruction at a time, to locate investigate the true root cause.

1. From the **Home** ribbon use the **Step Into Back** command to step back three instructions. As you do this, continue to examine the stack and memory windows.

   The command window will display the time travel position and the registers as you step back three instructions.

```
0:000> t-
Time Travel Position: 67:40
eax=00000000 ebx=00cf8000 ecx=99da9203 edx=69cf1a6c esi=00191046 edi=00191046
eip=00540020 esp=00effe4c ebp=00520055 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
00540020 ??              ???

0:000> t-
Time Travel Position: 67:3F
eax=00000000 ebx=00cf8000 ecx=99da9203 edx=69cf1a6c esi=00191046 edi=00191046
eip=0019193d esp=00effe48 ebp=00520055 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
DisplayGreeting!main+0x4d:
0019193d c3

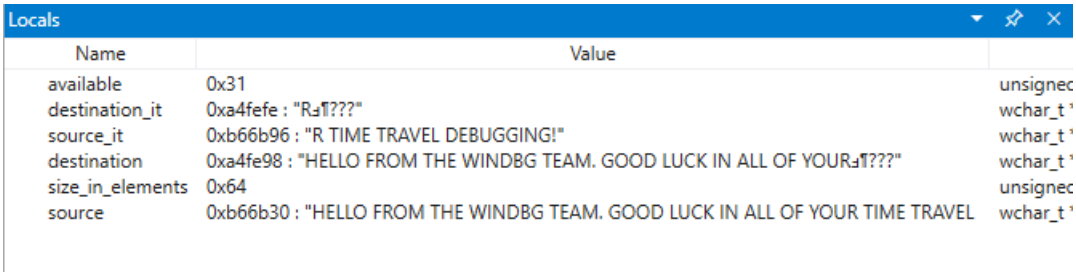
0:000> t-
Time Travel Position: 67:39
eax=0000004c ebx=00cf8000 ecx=99da9203 edx=69cf1a6c esi=00191046 edi=00191046
eip=00191935 esp=00effd94 ebp=00effe44 iopl=0         nv up ei pl nz ac po nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000212
DisplayGreeting!main+0x45:
```

> **NOTE**
> In this walkthrough, the command output shows the commands that can be used instead of the UI Menu options to allow users with a command line usage preference to use command line commands.

2. At this point in the trace our stack and base pointer have values that make more sense, so it appears that we have getting closer to the point in the code where the corruption occurred.

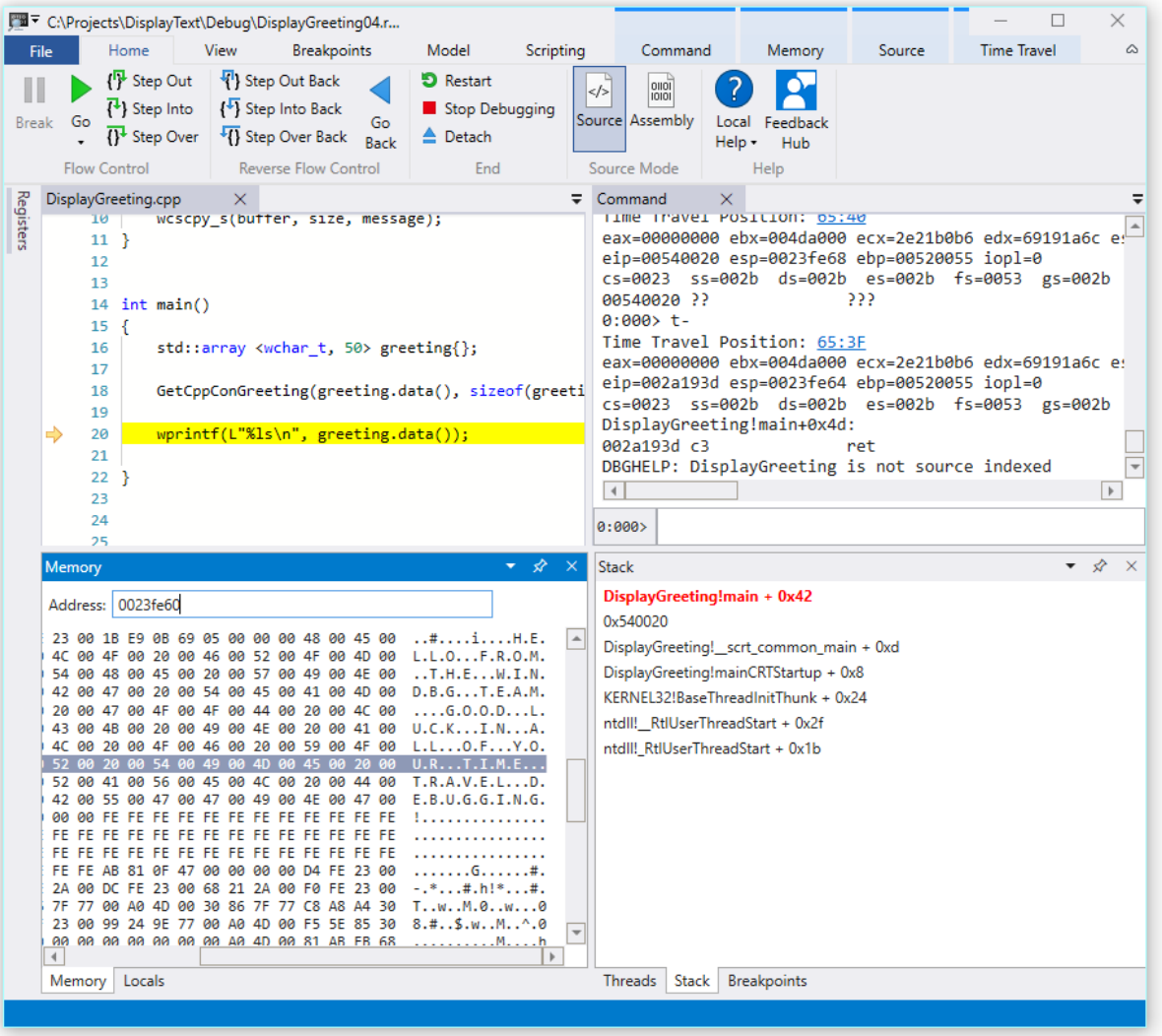
```
esp=00effd94 ebp=00effe44
```

   Also of interest is that the locals window contains values from our target app and the source code window is highlighting the line of code that is ready to be executed at this point in the trace.



3. To further investigate, we can open up a memory window to view the contents near the base pointer memory address of *0x00effe44*.

4. To display the associated ASCII characters, from the Memory ribbon, select **Text** and then **ASCII**.



5. Instead of the base pointer pointing to an instruction it is pointing to our message text. So something is not right here, this may be close to the point in time that we have corrupted the stack. To further investigate we will set a breakpoint.

> **NOTE**
>
> In this very small sample it would be pretty easy to just look in the code, but if there are hundreds of lines of code and dozens of subroutines the techniques described here can be used to decrease the time necessary to locate the issue.

TTD and breakpoints

Using breakpoints is a common approach to pause code execution at some event of interest. TTD allows you to set a breakpoint and travel back in time until that breakpoint is hit after the trace has been recorded. The ability to examine the process state after an issue has happened, to determine the best location for a breakpoint, enables additional debugging workflows unique to TTD.

Memory access breakpoints

You can set breakpoints that fire when a memory location is accessed. Use the **ba** (break on access) command, with the following syntax.

```
ba <access> <size> <address> {options}
```

OPTION	DESCRIPTION
e	execute (when CPU fetches an instruction from the address)
r	read/write (when CPU reads or writes to the address)
w	write (when the CPU writes to the address)

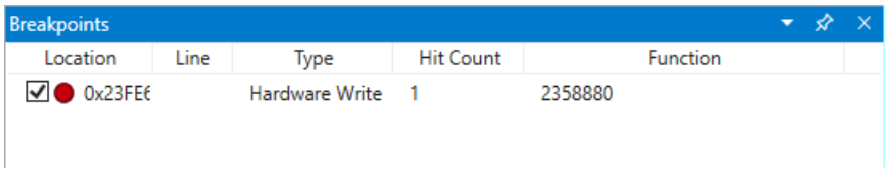
Note that you can only set four data breakpoints at any given time and it is up to you to make sure that you are aligning your data correctly or you won't trigger the breakpoint (words must end in addresses divisible by 2, dwords must be divisible by 4, and quadwords by 0 or 8).

Set the break on memory access breakpoint for the base pointer

1. At this point in the trace we would like to set a breakpoint on write memory access to base pointer - ebp which in our example is 00effe44. To do this use the **ba** command using the address we want to monitor. We want to monitor writes for four bytes, so we specify w4.

```
0:000> ba w4 00effe44
```

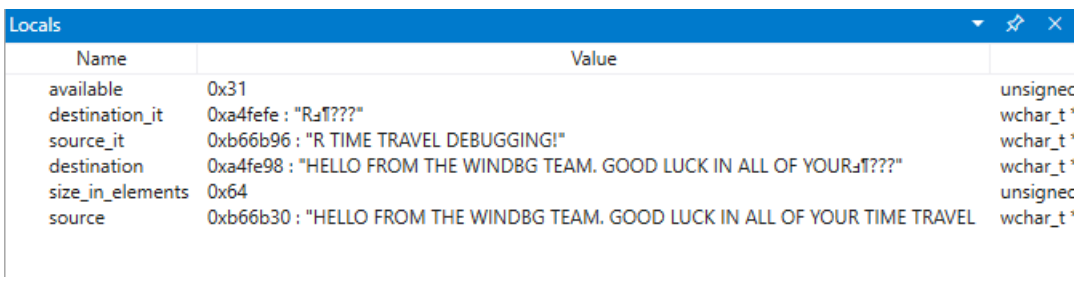
2. Select **View** and then **Breakpoints** to confirm they are set as intended.



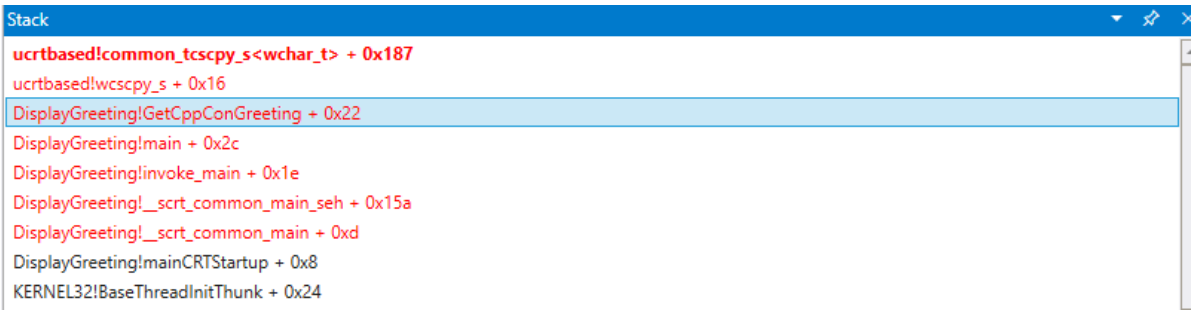
3. From the Home menu, select **Go Back** to travel back in time until the breakpoint is hit.

```
0:000> g-
Breakpoint 0 hit
Time Travel Position: 5B:92
eax=0000000f ebx=003db000 ecx=00000000 edx=00cc1a6c esi=00d41046 edi=0053fde8
eip=00d4174a esp=0053fcf8 ebp=0053fde8 iopl=0         nv up ei pl nz ac pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000216
DisplayGreeting!DisplayGreeting+0x3a:
00d4174a c745e000000000  mov     dword ptr [ebp-20h],0 ss:002b:0053fdc8=cccccccc
```

4. Select **View** and then **Locals**. In the locals window we can see that the *destination* variable has only part of the message, while the *source* has contains all of the text. This information supports the idea that the stack was corrupted.



5. At this point we can examine the program stack to see what code is active. From the **View** ribbon select **Stack**.



As it is very unlikely that the Microsoft provided wscpy_s() function would have a code bug like this, we look further in the stack. The stack shows that Greeting!main calls Greeting!GetCppConGreeting. In our very small code sample we could just open the code at this point and likely find the error pretty easily. But to illustrate the techniques that can be used with larger, more complex program, we will set a new breakpoint to investigate further.

Set the break on access breakpoint for the GetCppConGreeting function

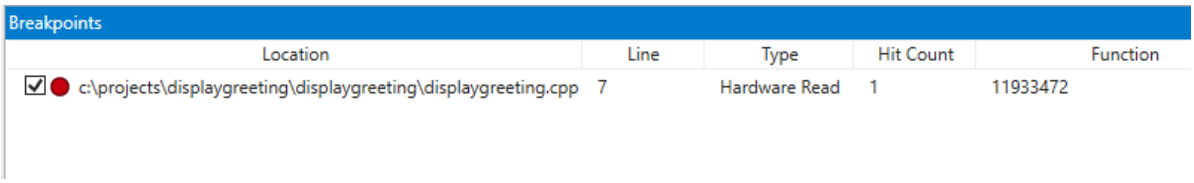
1. Use the breakpoints window to clear the existing breakpoint by right clicking on the existing breakpoint and selecting **Remove**.
2. Determine the address of the DisplayGreeting!DetermineStringSize function using the **dx** command.

```
0:000> dx &DisplayGreeting!GetCppConGreeting
&DisplayGreeting!GetCppConGreeting                : 0xb61720 [Type: void (__cdecl*)(wchar_t *,unsigned
int)]
    [Type: void __cdecl(wchar_t *,unsigned int)]
```

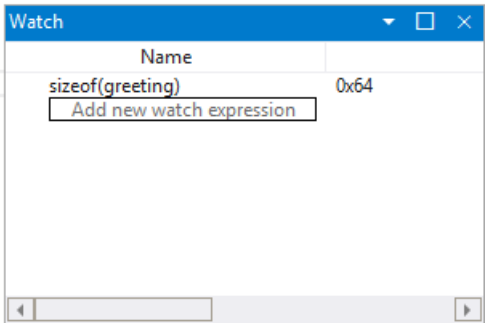
3. Use the **ba** command to set a breakpoint on memory access. Because the function will just be read from memory for execution, we need to set a r - read breakpoint.

```
0:000> ba r4 b61720
```

4. Confirm that a Hardware Read breakpoint is active in the breakpoints window.



5. As we are wondering about the size of the greeting string we will set a watch window to display the value of sizeof(greeting). From the View ribbon, select **Watch** and provide *sizeof(greeting)*.



6. On the Time Travel menu, use **Time travel to start** command to move to the start of the trace.

```
0:000> !tt 0
Setting position to the beginning of the trace
Setting position: 15:0
(1e5c.710): Break instruction exception - code 80000003 (first/second chance not available)
Time Travel Position: 15:0
eax=68e28100 ebx=00000000 ecx=77a266ac edx=69e34afc esi=69e3137c edi=00fa2000
eip=77a266ac esp=00ddf3b8 ebp=00ddf608 iopl=0         nv up ei pl nz na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000206
ntdll!LdrpInitializeProcess+0x1d1c:
77a266ac 83bdbcfeffff00  cmp     dword ptr [ebp-144h],0 ss:002b:00ddf4c4=00000000
```

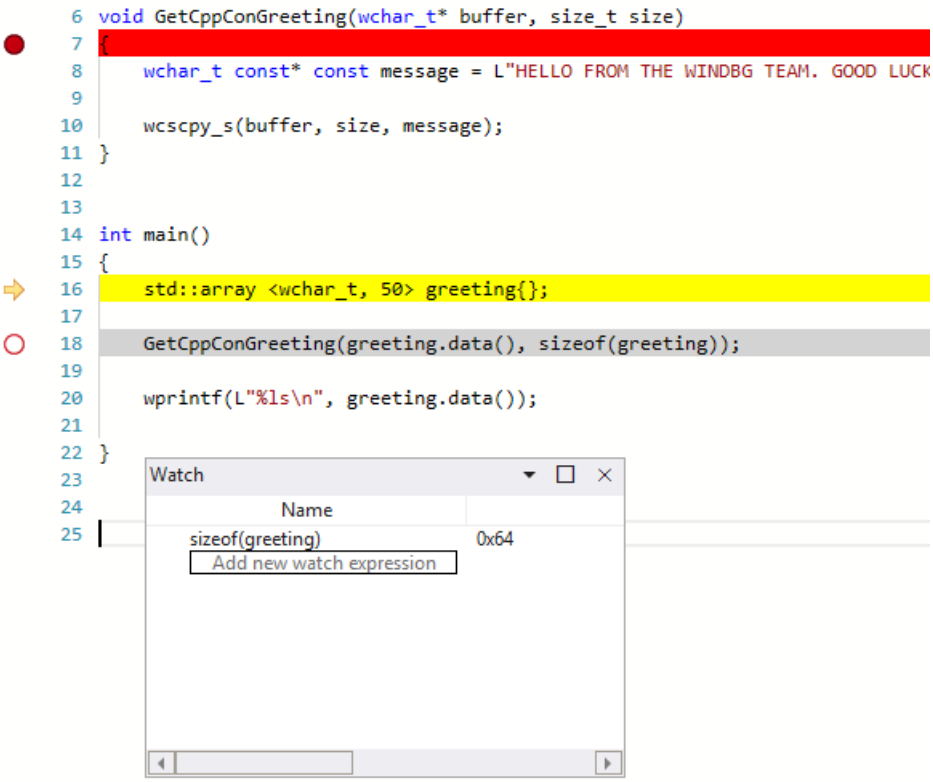
7. On the Home menu, select **Go** to move forward in the code until the breakpoint is hit.

```
0:000> g
Breakpoint 2 hit
Time Travel Position: 4B:1AD
eax=00ddf800 ebx=00fa2000 ecx=00ddf800 edx=00b61046 esi=00b61046 edi=00b61046
eip=00b61721 esp=00ddf7a4 ebp=00ddf864 iopl=0         nv up ei pl nz na po nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000202
DisplayGreeting!GetCppConGreeting+0x1:
00b61721 8bec            mov     ebp,esp
```

8. On the Home menu, select **Step Out Back** to back one step.

```
0:000> g-u
Time Travel Position: 4B:1AA
eax=00ddf800 ebx=00fa2000 ecx=00ddf800 edx=00b61046 esi=00b61046 edi=00b61046
eip=00b61917 esp=00ddf7ac ebp=00ddf864 iopl=0         nv up ei pl nz na po nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000202
DisplayGreeting!main+0x27:
00b61917 e8def7ffff      call    DisplayGreeting!ILT+245(?GetCppConGreetingYAXPA_WIZ) (00b610fa)
```

9. It looks like we have found the root cause. The *greeting* array that we declared is 50 characters in length, while the sizeof(greeting) that we pass into GetCppConGreeting is 0x64, 100).



As we look at the size issue further, we also notice that the message is 75 characters in length, 76 including the end of string character.

```
HELLO FROM THE WINDBG TEAM. GOOD LUCK IN ALL OF YOUR TIME TRAVEL DEBUGGING!
```

10. One way to fix the code would be to expand the size of the character array to 100.

```
std::array <wchar_t, 100> greeting{};
```

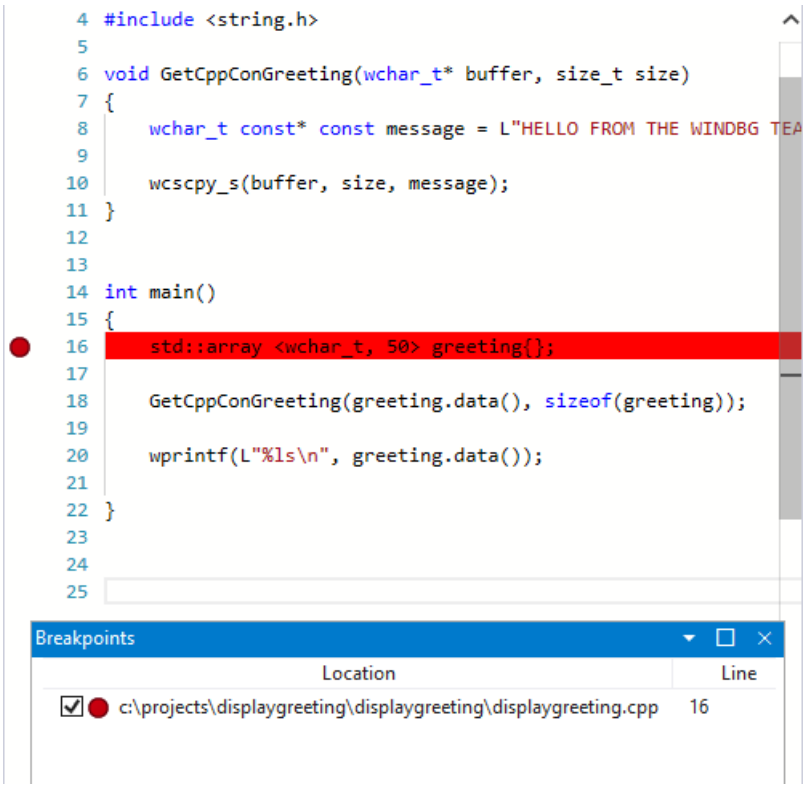
And we also need to change sizeof(greeting) to size(greeting) in this line of code.

```
 GetCppConGreeting(greeting.data(), size(greeting));
```

11. To validate these fixes, we could recompile the code and confirm that it runs without error.

Setting a breakpoint using the source window

1. An alternative way to perform this investigation would be to set a breakpoint by clicking on any line of code. For example clicking on the right side of the std:array definition line in the source window will set a breakpoint there.



2. On the Time Travel menu, use **Time travel to start** command to move to the start of the trace.

```
0:000> !tt 0
Setting position to the beginning of the trace
Setting position: 15:0
(1e5c.710): Break instruction exception - code 80000003 (first/second chance not available)
Time Travel Position: 15:0
eax=68e28100 ebx=00000000 ecx=77a266ac edx=69e34afc esi=69e3137c edi=00fa2000
eip=77a266ac esp=00ddf3b8 ebp=00ddf608 iopl=0         nv up ei pl nz na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000206
ntdll!LdrpInitializeProcess+0x1d1c:
77a266ac 83bdbcfeffff00  cmp     dword ptr [ebp-144h],0 ss:002b:00ddf4c4=00000000
```

3. On the Home Ribbon click on **Go** to travel back until the breakpoint is hit.

```
Breakpoint 0 hit
Time Travel Position: 5B:AF
eax=0000000f ebx=00c20000 ecx=00000000 edx=00000000 esi=013a1046 edi=00effa60
eip=013a17c1 esp=00eff970 ebp=00effa60 iopl=0         nv up ei pl nz na po nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000202
DisplayGreeting!DisplayGreeting+0x41:
013a17c1 8bf4            mov     esi,esp
```

Set the break on access breakpoint for the *greeting* variable

Another alternative way to perform this investigation, would be to set a breakpoint on suspect variables and examine what code is changing them. For example, to set a breakpoint on the greeting variable in the GetCppConGreeting method, use this procedure.

This portion of the walkthrough assumes that you are still located at the breakpoint from the previous section.

1. From **View** and then **Locals**. In the locals window, *greeting* is available in the current context, so we will be able to determine its memory location.

2. Use the **dx** command to examine the *greeting* array.

```
0:000> dx &greeting
&greeting                 : 0xddf800 [Type: std::array<wchar_t,50> *]
   [+0x000] _Elems           : "꽘棶棚瞝???" [Type: wchar_t [50]]
```

   In this trace, *greeting* is located in memory at ddf800.

3. Use the breakpoints window to clear any existing breakpoint by right clicking on the existing breakpoint and selecting **Remove**.

4. Set the breakpoint with the **ba** command using the memory address we want to monitor for write access.

```
ba w4 ddf800
```

5. On the Time Travel menu, use **Time travel to start** command to move to the start of the trace.

```
0:000> !tt 0
Setting position to the beginning of the trace
Setting position: 15:0
(1e5c.710): Break instruction exception - code 80000003 (first/second chance not available)
Time Travel Position: 15:0
eax=68e28100 ebx=00000000 ecx=77a266ac edx=69e34afc esi=69e3137c edi=00fa2000
eip=77a266ac esp=00ddf3b8 ebp=00ddf608 iopl=0         nv up ei pl nz na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000206
ntdll!LdrpInitializeProcess+0x1d1c:
77a266ac 83bdbcfefffff00  cmp     dword ptr [ebp-144h],0 ss:002b:00ddf4c4=00000000
```

6. On the Home menu, select **Go** to travel forward to the first point of memory access of the greeting array.

```
0:000> g-
Breakpoint 0 hit
Time Travel Position: 5B:9C
eax=cccccccc ebx=002b1000 ecx=00000000 edx=68d51a6c esi=013a1046 edi=001bf7d8
eip=013a1735 esp=001bf6b8 ebp=001bf7d8 iopl=0         nv up ei pl nz na po nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000202
DisplayGreeting!DetermineStringSize+0x25:
013a1735 c745ec04000000  mov     dword ptr [ebp-14h],4 ss:002b:001bf7c4=cccccccc
```

Alternatively, we could have traveled to the end of the trace and worked in reverse through the code to find that last point in the trace that the array memory location was written to.

Use the TTD.Memory objects to view memory access

Another way to determine at what points in the trace memory has been accessed, is to use the TTD.Memory objects and the dx command.

1. Use the **dx** command to examine the *greeting* array.

```
0:000> dx &greeting
&greeting                 : 0xddf800 [Type: std::array<wchar_t,50> *]
   [+0x000] _Elems           : "꽘棶棚瞝???" [Type: wchar_t [50]]
```

   In this trace, *greeting* is located in memory at ddf800.

2. Use the **dx** command to look at the four bytes in memory starting at that address with the read write access.

```
0:000> dx -r1 @$cursession.TTD.Memory(0xddf800,0xddf804, "rw")
@$cursession.TTD.Memory(0x1bf7d0,0x1bf7d4, "rw")
    [0x0]
    [0x1]
    [0x2]
    [0x3]
    [0x4]
    [0x5]
    [0x6]
    [0x7]
    [0x8]
    [0x9]
    [0xa]
    ...
```

3. Click on any of the occurrences to display more information about that occurrence of memory access.

```
0:000> dx -r1 @$cursession.TTD.Memory(0xddf800,0xddf804, "rw")[5]
@$cursession.TTD.Memory(0xddf800,0xddf804, "rw")[5]
    EventType        : MemoryAccess
    ThreadId         : 0x710
    UniqueThreadId   : 0x2
    TimeStart        : 27:3C1 [Time Travel]
    TimeEnd          : 27:3C1 [Time Travel]
    AccessType       : Write
    IP               : 0x6900432f
    Address          : 0xddf800
    Size             : 0x4
    Value            : 0xddf818
```

4. Click on [Time Travel] to position the trace at the point in time.

```
0:000> dx @$cursession.TTD.Memory(0xddf800,0xddf804, "rw")[5].TimeStart.SeekTo()
@$cursession.TTD.Memory(0xddf800,0xddf804, "rw")[5].TimeStart.SeekTo()
(1e5c.710): Break instruction exception - code 80000003 (first/second chance not available)
Time Travel Position: 27:3C1
eax=00ddf81c ebx=00fa2000 ecx=00ddf818 edx=ffffffff esi=00000000 edi=00b61046
eip=6900432f esp=00ddf804 ebp=00ddf810 iopl=0         nv up ei pl nz ac po nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000212
ucrtbased!_register_onexit_function+0xf:
6900432f 51              push    ecx
```

5. If we are interested in the last occurrence of read/write memory access in the trace we can click on the last item in the list or append the .Last() function to the end of the dx command.

```
0:000> dx -r1 @$cursession.TTD.Memory(0xddf800,0xddf804, "rw").Last()
@$cursession.TTD.Memory(0xddf800,0xddf804, "rw").Last()
    EventType        : MemoryAccess
    ThreadId         : 0x710
    UniqueThreadId   : 0x2
    TimeStart        : 53:100E [Time Travel]
    TimeEnd          : 53:100E [Time Travel]
    AccessType       : Read
    IP               : 0x690338e4
    Address          : 0xddf802
    Size             : 0x2
    Value            : 0x45
```

6. We could then click on [Time Travel] to move to that position in the trace and look further at the code execution at that point, using the techniques described earlier in this lab.

For more information about the TTD.Memory objects, see TTD.Memory Object.

Summary

In this very small sample the issue could have been determined by looking at the few lines of code, but in larger programs the techniques presented here can be used to decrease the time necessary to locate an issue.

Once a trace is recorded, the trace and repro steps can be shared, and the issue will be reproducible on demand on any PC.

See Also

Time Travel Debugging - Overview

Time Travel Debugging - Recording

Time Travel Debugging - Replay a trace

Time Travel Debugging - Working with trace files

Introduction to Time Travel Debugging objects

10/25/2019 • 14 minutes to read • Edit Online

This section describes how to use the data model to query time travel traces. This can be a powerful tool to answer questions like these about the code that is captured in a time travel trace.

- What exceptions are in the trace?
- At what point in time in the trace did a specific code module load?
- When were threads created/terminated in the trace?
- What are the longest running threads in the trace?

There are TTD extensions that add data to the *Session* and *Process* data model objects. The TTD data model objects can be accessed through the dx (Display Debugger Object Model Expression) command, WinDbg Preview's model windows, JavaScript and C++. The TTD extensions are automatically loaded when debugging a time travel trace.

Process Objects

The primary objects added to *Process* objects can be found in the *TTD* namespace off of any *Process* object. For example, `@$curprocess.TTD`.

```
:000> dx @$curprocess.TTD
@$curprocess.TTD
    Threads
    Events
    Lifetime         : [26:0, 464232:0]
    SetPosition      [Sets the debugger to point to the given position on this process.]
```

For general information on working with LINQ queries and debugger objects, see Using LINQ With the debugger objects.

Properties

OBJECT	DESCRIPTION
Lifetime	A TTD range object describing the lifetime of the entire trace.
Threads	Contains a collection of TTD thread objects, one for every thread throughout the lifetime of the trace.
Events	Contains a collection of TTD event objects, one for every event in the trace.

Methods

METHOD	DESCRIPTION
SetPosition()	Takes an integer between 0 and 100 or string in N:N form as input and jumps the trace to that location. See !tt for more information.

Session Objects

The primary objects added to *Session* objects can be found in the *TTD* namespace off of any *Session* object. For example, `@$cursession.TTD`.

```
0:000> dx @$cursession.TTD
@$cursession.TTD                 : [object Object]
    Calls            [Returns call information from the trace for the specified set of methods:
TTD.Calls("module!method1", "module!method2", ...) For example: dx
@$cursession.TTD.Calls("user32!SendMessageA")]
    Memory           [Returns memory access information for specified address range: TTD.Memory(startAddress,
endAddress [, "rwec"])]
    DefaultParameterCount : 0x4
    AsyncQueryEnabled : false
    Resources
    Data             : Normalized data sources based on the contents of the time travel trace
    Utility          : Methods that can be useful when analyzing time travel traces
```

> **NOTE**
> There are some objects and methods added by TTDAnalyze that are used for internal functions of the extension. Not all namespaces are documented, and the current namespaces will evolve over time.

Methods

METHOD	DESCRIPTION
Data.Heap()	A collection of heap objects that were allocated during the trace. Note that this is a function that does computation, so it takes a while to run.
Calls()	Returns a collection of calls objects that match the input string. The input string can contain wildcards. Note that this is a function that does computation, so it takes a while to run.
Memory()	This is a method that takes beginAddress, endAddress and dataAccessMask parameters and returns a collection of memory objects. Note that this is a function that does computation, so it takes a while to run.

Sorting query output

Use the OrderBy() method to sort the rows returned from the query by one or more columns. This example sorts by TimeStart in ascending order.

```
0:000> dx -r2 @$cursession.TTD.Calls("kernelbase!GetLastError").OrderBy(c => c.TimeStart)
@$cursession.TTD.Calls("kernelbase!GetLastError").OrderBy(c => c.TimeStart)
    [0xb]
        EventType        : Call
        ThreadId         : 0x3a10
        UniqueThreadId   : 0x2
        TimeStart        : 39:2DC [Time Travel]
        TimeEnd          : 39:2DF [Time Travel]
        Function         : UnknownOrMissingSymbols
        FunctionAddress  : 0x7561ccc0
        ReturnAddress    : 0x7593d24c
        ReturnValue      : 0x0
        Parameters
    [0xe]
        EventType        : Call
        ThreadId         : 0x3a10
        UniqueThreadId   : 0x2
        TimeStart        : AF:36 [Time Travel]
        TimeEnd          : AF:39 [Time Travel]
        Function         : UnknownOrMissingSymbols
        FunctionAddress  : 0x7561ccc0
        ReturnAddress    : 0x4723ef
        ReturnValue      : 0x0
        Parameters
```

To display additional depth of the data model objects the -r2 recursion level option is used. For more information about the dx command options, see dx (Display Debugger Object Model Expression).

This example sorts by TimeStart in descending order.

```
0:000> dx -r2 @$cursession.TTD.Calls("kernelbase!GetLastError").OrderByDescending(c => c.TimeStart)
@$cursession.TTD.Calls("kernelbase!GetLastError").OrderByDescending(c => c.TimeStart)
    [0x1896]
        EventType        : Call
        ThreadId         : 0x3a10
        UniqueThreadId   : 0x2
        TimeStart        : 464224:34 [Time Travel]
        TimeEnd          : 464224:37 [Time Travel]
        Function         : UnknownOrMissingSymbols
        FunctionAddress  : 0x7561ccc0
        ReturnAddress    : 0x7594781c
        ReturnValue      : 0x0
        Parameters
    [0x18a0]
        EventType        : Call
        ThreadId         : 0x3a10
        UniqueThreadId   : 0x2
        TimeStart        : 464223:21 [Time Travel]
        TimeEnd          : 464223:24 [Time Travel]
        Function         : UnknownOrMissingSymbols
        FunctionAddress  : 0x7561ccc0
        ReturnAddress    : 0x7594781c
        ReturnValue      : 0x0
        Parameters
```

Specifying elements in a query

To select a specific element a variety of qualifiers can be appended to the query. For example, the query displays the first call that contains "kernelbase!GetLastError".

```
0:000> dx @$cursession.TTD.Calls("kernelbase!GetLastError").First()
@$cursession.TTD.Calls("kernelbase!GetLastError").First()
    EventType        : Call
    ThreadId         : 0x3a10
    UniqueThreadId   : 0x2
    TimeStart        : 77A:9 [Time Travel]
    TimeEnd          : 77A:C [Time Travel]
    Function         : UnknownOrMissingSymbols
    FunctionAddress  : 0x7561ccc0
    ReturnAddress    : 0x6cf12406
    ReturnValue      : 0x0
    Parameters
```

Filtering in a query

Use the Select() method to choose which columns to see and modify the column display name.

This example returns rows where ReturnValue is not zero and selects to display the TimeStart and ReturnValue columns with custom display names of Time and Error.

```
0:000> dx -r2 @$cursession.TTD.Calls("kernelbase!GetLastError").Where(c => c.ReturnValue != 0).Select(c =>
new { Time = c.TimeStart, Error = c.ReturnValue })
@$cursession.TTD.Calls("kernelbase!GetLastError").Where(c => c.ReturnValue != 0).Select(c => new { Time =
c.TimeStart, Error = c.ReturnValue })
    [0x13]
        Time             : 3C64A:834 [Time Travel]
        Error            : 0x36b7
    [0x1c]
        Time             : 3B3E7:D6 [Time Travel]
        Error            : 0x3f0
    [0x1d]
        Time             : 3C666:857 [Time Travel]
        Error            : 0x36b7
    [0x20]
        Time             : 3C67E:12D [Time Travel]
```

Grouping

Use the GroupBy() method to group data returned by the query to do perform analysis using structured results This example groups the time locations by error number.

```
0:000> dx -r2 @$cursession.TTD.Calls("kernelbase!GetLastError").Where(c => c.ReturnValue != 0).Select(c =>
new { Time = c.TimeStart, Error = c.ReturnValue }).GroupBy(x => x.Error)
@$s = @$cursession.TTD.Calls("kernelbase!GetLastError").Where(c => c.ReturnValue != 0).Select(c => new { Time
= c.TimeStart, Error = c.ReturnValue }).GroupBy(x => x.Error)
    [0x36b7]
        [0x0]
        [0x1]
        [0x2]
        [0x3]
        [...]
    [0x3f0]
        [0x0]
        [0x1]
        [0x2]
        [0x3]
...
```

Assigning result of a query to a variable

Use this syntax to assigning result of a query to a variable `dx @$var = <expression>`

This example assigns the results of a query to myResults

```
dx -r2 @$myResults = @$cursession.TTD.Calls("kernelbase!GetLastError").Where(c => c.ReturnValue !=
0).Select(c => new { Time = c.TimeStart, Error = c.ReturnValue })
```

Use the dx command to display the newly created variable using the -g grid option. For more information on the dx command options, see dx (Display Debugger Object Model Expression).

```
0:000> dx -g @$myResults
=======================================
=           = (+) Time     = (+) Error =
=======================================
= [0x13]    - 3C64A:834    - 0x36b7    =
= [0x1c]    - 3B3E7:D6     - 0x3f0     =
= [0x1d]    - 3C666:857    - 0x36b7    =
= [0x20]    - 3C67E:12D    - 0x2       =
= [0x21]    - 3C6F1:127    - 0x2       =
= [0x23]    - 3A547:D6     - 0x3f0     =
= [0x24]    - 3A59B:D0     - 0x3f0     =
```

Examples

Querying for exceptions

This LINQ query uses the TTD.Event object to display all of the exceptions in the trace.

```
0:000> dx @$curprocess.TTD.Events.Where(t => t.Type == "Exception").Select(e => e.Exception)
@$curprocess.TTD.Events.Where(t => t.Type == "Exception").Select(e => e.Exception)
    [0x0]            : Exception 0x000006BA of type Software at PC: 0X777F51D0
    [0x1]            : Exception 0x000006BA of type Software at PC: 0X777F51D0
    [0x2]            : Exception 0xE06D7363 of type CPlusPlus at PC: 0X777F51D0
```

Querying for specific API calls

Use TTD.Calls object to query for specific API calls. In this example, an error has occurred when calling *user32!MessageBoxW*, the Windows API to show a message box. We list all calls to MessageBoxW , order it by the start time of the function, and then pick the last call.

```
0:000> dx @$cursession.TTD.Calls("user32!MessageBoxW").OrderBy(c => c.TimeStart).Last()
@$cursession.TTD.Calls("user32!MessageBoxW").OrderBy(c => c.TimeStart).Last()
    EventType        : Call
    ThreadId         : 0x3a10
    UniqueThreadId   : 0x2
    TimeStart        : 458310:539 [Time Travel]
    TimeEnd          : 45C648:61 [Time Travel]
    Function         : UnknownOrMissingSymbols
    FunctionAddress  : 0x750823a0
    ReturnAddress    : 0x40cb93
    ReturnValue      : 0x10a7000000000001
    Parameters
```

Querying for the load event of a specific module

First, use the lm (List Loaded Modules) command to display the loaded modules.

```
0:000> lm
start    end        module name
012b0000 012cf000   CDog_Console   (deferred)
11570000 1158c000   VCRUNTIME140D   (deferred)
11860000 119d1000   ucrtbased   (deferred)
119e0000 11b63000   TTDRecordCPU   (deferred)
11b70000 11cb1000   TTDWriter   (deferred)
73770000 73803000   apphelp    (deferred)
73ea0000 74062000   KERNELBASE   (deferred)
75900000 759d0000   KERNEL32   (deferred)
77070000 771fe000   ntdll      (private pdb symbols)
```

Then use the following dx command to see at what position in the trace a specific module was loaded, such as ntdll.

```
dx @$curprocess.TTD.Events.Where(t => t.Type == "ModuleLoaded").Where(t =>
t.Module.Name.Contains("ntdll.dll"))
@$curprocess.TTD.Events.Where(t => t.Type == "ModuleLoaded").Where(t => t.Module.Name.Contains("ntdll.dll"))
    [0x0]            : Module Loaded at position: A:0
```

This LINQ query displays the load event(s) of a particular module.

```
0:000> dx @$curprocess.TTD.Events.Where(t => t.Type == "ModuleUnloaded").Where(t =>
t.Module.Name.Contains("ntdll.dll"))
@$curprocess.TTD.Events.Where(t => t.Type == "ModuleUnloaded").Where(t =>
t.Module.Name.Contains("ntdll.dll"))
    [0x0]            : Module Unloaded at position: FFFFFFFFFFFFFFFE:0
```

The address of FFFFFFFFFFFFFFFE:0 indicates the end of the trace.

Querying for all of the error checks in the trace

Use this command to sort by all of the error checks in the trace by error count.

```
0:000> dx -g @$cursession.TTD.Calls("kernelbase!GetLastError").Where( x=> x.ReturnValue != 0).GroupBy(x =>
x.ReturnValue).Select(x => new { ErrorNumber = x.First().ReturnValue, ErrorCount =
x.Count()}).OrderByDescending(p => p.ErrorCount),d
================================================
=                 = (+) ErrorNumber = ErrorCount =
================================================
= [1008]          - 1008            - 8668       =
= [14007]         - 14007           - 4304       =
= [2]             - 2               - 1710       =
= [6]             - 6               - 1151       =
= [1400]          - 1400            - 385        =
= [87]            - 87              - 383        =
```

Querying for the time position in the trace when threads were created

Use this dx command to display all of the events in the trace in grid format (-g).

```
0:000> dx -g @$curprocess.TTD.Events
================================================================================================================================
================================================================================
=                                                    = (+) Type            = (+) Position          =
(+) Module                                             = (+) Thread             =
================================================================================================================================
================================================================================
= [0x0] : Module Loaded at position: 2:0                   - ModuleLoaded        - 2:0                   -
Module C:\Users\USER1\Documents\Visual Studio 2015\Proje... -                        =
= [0x1] : Module Loaded at position: 3:0                   - ModuleLoaded        - 3:0                   -
Module C:\WINDOWS\SYSTEM32\VCRUNTIME140D.dll at address 0... -                          =
= [0x2] : Module Loaded at position: 4:0                   - ModuleLoaded        - 4:0                   -
Module C:\WINDOWS\SYSTEM32\ucrtbased.dll at address 0X118... -                         =
= [0x3] : Module Loaded at position: 5:0                   - ModuleLoaded        - 5:0                   -
Module C:\Users\USER1\AppData\Local\Dbg\UI\Fast.20170907... -                        =
= [0x4] : Module Loaded at position: 6:0                   - ModuleLoaded        - 6:0                   -
Module C:\Users\USER1\AppData\Local\Dbg\UI\Fast.20170907... -                        =
= [0x5] : Module Loaded at position: 7:0                   - ModuleLoaded        - 7:0                   -
Module C:\WINDOWS\SYSTEM32\apphelp.dll at address 0X73770... -                          =
= [0x6] : Module Loaded at position: 8:0                   - ModuleLoaded        - 8:0                   -
Module C:\WINDOWS\System32\KERNELBASE.dll at address 0X73... -                          =
= [0x7] : Module Loaded at position: 9:0                   - ModuleLoaded        - 9:0                   -
Module C:\WINDOWS\System32\KERNEL32.DLL at address 0X7590... -                          =
= [0x8] : Module Loaded at position: A:0                   - ModuleLoaded        - A:0                   -
Module C:\WINDOWS\SYSTEM32\ntdll.dll at address 0X7707000... -                          =
= [0x9] : Thread created at D:0                            - ThreadCreated       - D:0                   -
- UID: 2, TID: 0x4C2C    =
= [0xa] : Thread terminated at 64:0                        - ThreadTerminated    - 64:0                  -
- UID: 2, TID: 0x4C2C    =
= [0xb] : Thread created at 69:0                           - ThreadCreated       - 69:0                  -
- UID: 3, TID: 0x4CFC    =
= [0xc] : Thread created at 6A:0                           - ThreadCreated       - 6A:0                  -
- UID: 4, TID: 0x27B0    =
= [0xd] : Thread terminated at 89:0                        - ThreadTerminated    - 89:0                  -
- UID: 4, TID: 0x27B0    =
= [0xe] : Thread terminated at 8A:0                        - ThreadTerminated    - 8A:0                  -
- UID: 3, TID: 0x4CFC    =
= [0xf] : Module Unloaded at position: FFFFFFFFFFFFFFFE:0  - ModuleUnloaded      - FFFFFFFFFFFFFFFE:0    -
Module C:\Users\USER1\Documents\Visual Studio 2015\Proje... -                           =
= [0x10] : Module Unloaded at position: FFFFFFFFFFFFFFFE:0 - ModuleUnloaded      - FFFFFFFFFFFFFFFE:0    -
Module C:\Users\USER1\AppData\Local\Dbg\UI\Fast.20170907... -                           =
= [0x11] : Module Unloaded at position: FFFFFFFFFFFFFFFE:0 - ModuleUnloaded      - FFFFFFFFFFFFFFFE:0    -
Module C:\WINDOWS\SYSTEM32\VCRUNTIME140D.dll at address 0... -                          =
= [0x12] : Module Unloaded at position: FFFFFFFFFFFFFFFE:0 - ModuleUnloaded      - FFFFFFFFFFFFFFFE:0    -
Module C:\Users\USER1\AppData\Local\Dbg\UI\Fast.20170907... -                           =
= [0x13] : Module Unloaded at position: FFFFFFFFFFFFFFFE:0 - ModuleUnloaded      - FFFFFFFFFFFFFFFE:0    -
Module C:\WINDOWS\SYSTEM32\ucrtbased.dll at address 0X118... -                         =
= [0x14] : Module Unloaded at position: FFFFFFFFFFFFFFFE:0 - ModuleUnloaded      - FFFFFFFFFFFFFFFE:0    -
Module C:\WINDOWS\SYSTEM32\apphelp.dll at address 0X73770... -                          =
= [0x15] : Module Unloaded at position: FFFFFFFFFFFFFFFE:0 - ModuleUnloaded      - FFFFFFFFFFFFFFFE:0    -
Module C:\WINDOWS\System32\KERNELBASE.dll at address 0X73... -                          =
= [0x16] : Module Unloaded at position: FFFFFFFFFFFFFFFE:0 - ModuleUnloaded      - FFFFFFFFFFFFFFFE:0    -
Module C:\WINDOWS\System32\KERNEL32.DLL at address 0X7590... -                          =
= [0x17] : Module Unloaded at position: FFFFFFFFFFFFFFFE:0 - ModuleUnloaded      - FFFFFFFFFFFFFFFE:0    -
Module C:\WINDOWS\SYSTEM32\ntdll.dll at address 0X7707000... -                          =
================================================================================================================================
================================================================================
```

Click on any of the columns with a + sign to sort the output.

Use this LINQ query to display in grid format, the time position in the trace when threads were created (Type == "ThreadCreated").

```
dx -g @$curprocess.TTD.Events.Where(t => t.Type == "ThreadCreated").Select(t => t.Thread)
================================================================================================================
=                             = (+) UniqueId = (+) Id    = (+) Lifetime                 = (+) ActiveTime  =
================================================================================================================
= [0x0] : UID: 2, TID: 0x4C2C - 0x2          - 0x4c2c    - [0:0, FFFFFFFFFFFFFFFE:0]    - [D:0, 64:0]     =
= [0x1] : UID: 3, TID: 0x4CFC - 0x3          - 0x4cfc    - [0:0, 8A:0]                  - [69:0, 8A:0]    =
= [0x2] : UID: 4, TID: 0x27B0 - 0x4          - 0x27b0    - [0:0, 89:0]                  - [6A:0, 89:0]    =
================================================================================================================
```

Use this LINQ query to display in grid format, the time positions in the trace when threads were terminated (Type == "ThreadTerminated").

```
0:000> dx -g @$curprocess.TTD.Events.Where(t => t.Type == "ThreadTerminated").Select(t => t.Thread)
================================================================================================================
=                             = (+) UniqueId = (+) Id    = (+) Lifetime                 = (+) ActiveTime  =
================================================================================================================
= [0x0] : UID: 2, TID: 0x4C2C - 0x2          - 0x4c2c    - [0:0, FFFFFFFFFFFFFFFE:0]    - [D:0, 64:0]     =
= [0x1] : UID: 4, TID: 0x27B0 - 0x4          - 0x27b0    - [0:0, 89:0]                  - [6A:0, 89:0]    =
= [0x2] : UID: 3, TID: 0x4CFC - 0x3          - 0x4cfc    - [0:0, 8A:0]                  - [69:0, 8A:0]    =
================================================================================================================
```

Sorting output to determine the longest running threads

Use this LINQ query to display in grid format, the approximate longest running threads in the trace.

```
0:000> dx -g @$curprocess.TTD.Events.Where(e => e.Type == "ThreadTerminated").Select(e => new { Thread =
e.Thread, ActiveTimeLength = e.Thread.ActiveTime.MaxPosition.Sequence -
e.Thread.ActiveTime.MinPosition.Sequence }).OrderByDescending(t => t.ActiveTimeLength)
=========================================================
=          = (+) Thread              = ActiveTimeLength =
=========================================================
= [0x0]    - UID: 2, TID: 0x1750     - 0x364030         =
= [0x1]    - UID: 3, TID: 0x420C     - 0x360fd4         =
= [0x2]    - UID: 7, TID: 0x352C     - 0x35da46         =
= [0x3]    - UID: 9, TID: 0x39F4     - 0x34a5b5         =
= [0x4]    - UID: 11, TID: 0x4288    - 0x326199         =
= [0x5]    - UID: 13, TID: 0x21C8    - 0x2fa8d8         =
= [0x6]    - UID: 14, TID: 0x2188    - 0x2a03e3         =
= [0x7]    - UID: 15, TID: 0x40E8    - 0x29e7d0         =
= [0x8]    - UID: 16, TID: 0x124     - 0x299677         =
= [0x9]    - UID: 4, TID: 0x2D74     - 0x250f43         =
= [0xa]    - UID: 5, TID: 0x2DC8     - 0x24f921         =
= [0xb]    - UID: 6, TID: 0x3B1C     - 0x24ec8e         =
= [0xc]    - UID: 10, TID: 0x3808    - 0xf916f          =
= [0xd]    - UID: 12, TID: 0x26B8    - 0x1ed3a          =
= [0xe]    - UID: 17, TID: 0x37D8    - 0xc65            =
= [0xf]    - UID: 8, TID: 0x45F8     - 0x1a2            =
=========================================================
```

Querying for read accesses to a memory range

Use the TTD.Memory object to query for to query for read accesses to a memory range.

The Thread Environment Block (TEB) is a structure that contains all the information regarding the state of a thread, including the result returned by GetLastError(). You can query this data structure by running `dx @$teb` for the current thread. One of TEB's members is a LastErrorValue variable, 4 bytes in size. We can reference the LastErrorValue member in the TEB using this syntax. `dx &@$teb->LastErrorValue`.

The example query shows how to find every read operation done in that range in memory, select all the reads that happen before the a dialog was created and then sort the result to find the last read operation.

```
0:000> dx @$cursession.TTD.Memory(&@$teb->LastErrorValue, &@$teb->LastErrorValue + 0x4, "r")
@$cursession.TTD.Memory(&@$teb->LastErrorValue, &@$teb->LastErrorValue + 0x4, "r")
    [0x0]
    [0x1]
    [0x2]
    [0x3]
```

If in our trace a "dialog" event has taken place we can run a query to find every read operation done in that range in memory, select all the reads that happen before the dialog got created and then sort the result to find the last read operation. Then time travel to that point in time by calling SeekTo() on the resulting time position.

```
:000> dx @$cursession.TTD.Memory(&@$teb->LastErrorValue, &@$teb->LastErrorValue + 0x4, "r").Where(m =>
m.TimeStart < @$dialog).OrderBy(m => m.TimeStart).Last().TimeEnd.SeekTo()
Setting position: 458300:37
ModLoad: 6cee0000 6cf5b000   C:\WINDOWS\system32\uxtheme.dll
ModLoad: 75250000 752e6000   C:\WINDOWS\System32\OLEAUT32.dll
ModLoad: 76320000 7645d000   C:\WINDOWS\System32\MSCTF.dll
ModLoad: 76cc0000 76cce000   C:\WINDOWS\System32\MSASN1.dll
```

GitHub TTD Query Lab

For a tutorial on how to debug C++ code using a Time Travel Debugging recording using queries to find information about the execution of the problematic code in question, see https://github.com/Microsoft/WinDbg-Samples/blob/master/TTDQueries/tutorial-instructions.md.

All of the code used in the lab is available here: https://github.com/Microsoft/WinDbg-Samples/tree/master/TTDQueries/app-sample.

Troubleshooting TTD Queries

"UnknownOrMissingSymbols" as the function names

The data model extension needs full symbol information in order to provide function names, parameter values, etc. When full symbol information is not available the debugger uses "UnknownOrMissingSymbols" as the function name.

- If you have private symbols you will get the function name and the correct list of parameters.
- If you have public symbols you will get the function name and a default set of parameters - four unsigned 64-bit ints.
- If you have no symbol information for the module you are querying, then “UnknownOrMissingSymbols” is used as the name.

TTD Queries for calls

There can be a several reasons that a query does not return anything for calls to a DLL.

- The syntax for the call isn't quite right. Try verifying the call syntax by using the x command: "x ". If the module name returned by x is in uppercase, use that.
- The DLL is not loaded yet and is loaded later in the trace. To work around this travel to a point in time after the DLL is loaded and redo the query.
- The call is inlined which the query engine is unable to track.
- The query pattern uses wildcards which returns too many functions. Try to make the query pattern more specific so that the number of matched functions is small enough.

See Also

Using LINQ With the debugger objects

dx (Display Debugger Object Model Expression)

Time Travel Debugging - Overview

Time Travel Debugging - JavaScript Automation

TTD Calls Objects

10/25/2019 • 2 minutes to read • Edit Online

Description

TTD Calls objects are used to give information about function calls that occur over the course of a trace.

Parameters

PROPERTY	DESCRIPTION
Function!SymbolName	One or more contained in double quotes, separated by a comma. For example dx @$cursession.TTD.Calls("module!symbol1", "module!symbol2", ...)

Properties

PROPERTY	DESCRIPTION
EventType	The type of event. This is "Call" for all TTD Calls objects.
ThreadId	The OS thread ID of thread that made the request.
UniqueThreadId	A unique ID for the thread across the trace. Regular thread IDs can get reused over the lifetime of a process but UniqueThreadIds cannot.
Function	The symbolic name of the function.
FunctionAddress	The function's address in memory.
ReturnValue	The return value of the function. If the function has a void type, this property will not be present.

Children

OBJECT	DESCRIPTION
Parameters[]	An array containing the parameters passed to the function. The number of elements varies based on the type signature of the function.
TimeStart	A position object that describes the position at the start of the call.
TimeEnd	A position object that describes the position at the end of the call.

Remarks

Time travel debugging uses symbol information provided in the PDBs to determine the number of parameters for a function and their types, the return value type, and the calling convention. In the event that symbol information is not available or the symbols have been restricted to public symbol information, it is still possible to do queries. The time travel query engine will make some assumptions in this scenario:

- There are four 64-bit unsigned integer parameters to the function
- The return value is a 64-bit unsigned integer
- The function name is set to a fixed string: "UnknownOrMissingSymbols"

These assumptions allow queries to be made in the absence of adequate symbol information. However, for best results use full PDB symbols when possible.

Note that the Calls function does computation, and depending on the size of the trace, it can take a while to run. CPU usage will spike during the computation and watching CPU usage in task manager, gives an indication that the computation is progressing. The query results are cached in memory so subsequent queries against previously queried calls are significantly faster.

Example Usage

This example shows the calls object for ucrtbase!initterm.

```
0:000> dx -r2 @$cursession.TTD.Calls("ucrtbase!initterm")
@$cursession.TTD.Calls("ucrtbase!initterm")
    [0x0]
        EventType        : Call
        ThreadId         : 0x2074
        UniqueThreadId   : 0x2
        TimeStart        : 1E:5D0
        TimeEnd          : 2D:E
        Function         : ucrtbase!_initterm
        FunctionAddress  : 0x7ffb345825d0
        ReturnAddress    : 0x7ff6a521677e
        Parameters
```

See Also

Time Travel Debugging - Introduction to Time Travel Debugging objects

Time Travel Debugging - Overview

TTD Heap Objects

10/25/2019 • 2 minutes to read • Edit Online

Description

TTD Heap objects are used to give information about heap calls that occur over the course of a trace.

Properties

Every heap object will have these properties.

PROPERTY	DESCRIPTION
Action	Describes the action that occurred. Possible values are: Alloc, ReAlloc, Free, Create, Protect, Lock, Unlock, Destroy.
Heap	The handle for the Win32 heap.

Conditional properties

Depending on the heap object, it may have some of the properties below.

PROPERTY	DESCRIPTION
Address	The address of the allocated object.
PreviousAddress	The address of the allocated object before it was reallocated. If Address is not the same as PreviousAddress then the reallocation caused the memory to move.
Size	The size and/or requested size of an allocated object.
BaseAddress	The address of an allocated object in the heap. It can represent the address which will be freed (Free) or the address of the object before it is reallocated (ReAlloc.)
Flags	Meaning depends on the API.
Result	The result of the heap API call. Non-zero means success and zero means failure.
ReserveSize	Amount of memory to reserve for the heap.
CommitSize	Initial committed size for the heap.
MakeReadOnly	A non-zero value indicates a request to make the heap read-only; A zero value indicates the heap should be read-write.

Children

OBJECT	DESCRIPTION
TimeStart	A position object that describes the position at the start of the allocation.
TimeEnd	A position object that describes the position at the end of the allocation.

Example Usage

Use this dx command to display the heap memory in a grid using the -g option.

```
0:0:000> dx -g @$cursession.TTD.Data.Heap()
================================================================================================================================
========================================
=                          = Action     = Heap          = Address       = Size      = Flags  = (+) TimeStart =
(+) TimeEnd = Result = PreviousAddress =
================================================================================================================================
========================================
= [0x0] : [object Object]  - Alloc      - 0xaf0000      - 0xb0cfd0      - 0x4c      - 0x0    - FAB:17B1      -
FAD:40      -        -                 =
= [0x1] : [object Object]  - Alloc      - 0xaf0000      - 0xb07210      - 0x34      - 0x8    - FB1:9         -
FB3:74      -        -                 =
= [0x2] : [object Object]  - Alloc      - 0xaf0000      - 0xb256d8      - 0x3c      - 0x8    - E525:174      -
E526:E1     -        -                 =
```

The output can be described as "normalized data" because there is a chosen set of APIs that represent heap operations. The data that is extracted from the appropriate parameters, is presented in a uniform manner.

Clicking on TimeStart or TimeEnd will navigate you to that point in the trace.

Click on the parameters field next to a specific entry, to display available parameter information.

```
dx -r1 @$cursession.TTD.Data.Heap()[2].@"Parameters"
@$cursession.TTD.Data.Heap()[2].@"Parameters"
    [0x0]            : 0x16c7d780000
    [0x1]            : 0x280000
    [0x2]            : 0x20
    [0x3]            : 0x0
```

See Also

Time Travel Debugging - Introduction to Time Travel Debugging objects

Time Travel Debugging - Overview

TTD Memory Objects

10/25/2019 • 3 minutes to read • Edit Online

Description

TTD Memory is a method that takes beginAddress, endAddress and dataAccessMask parameters and returns a collection of memory objects that contain memory access information.

Parameters

PROPERTY	DESCRIPTION
beginAddress	The beginning address of the memory object prefaced with 0x.
endAddress	The ending address of the memory object prefaced with 0x.
dataAccessMask	The data access mask contained in double quotes. This can be r for read, w for write, e for execute and c for change.

Children

OBJECT	DESCRIPTION
EventType	The type of event. This is "MemoryAccess" for all TTD.Memory objects.
ThreadId	The OS thread ID of thread that made the request.
UniqueThreadId	A unique ID for the thread across the trace. Regular thread IDs can get reused over the lifetime of a process but UniqueThreadIds cannot.
TimeStart	A position object that describes the position when memory access was made.
TimeEnd	A position object that describes the position when memory access was made. This will always be the same as the TimeStart for TTD.Memory objects.
AccessType	The access type - Read, Write or Execute.
IP	The instruction pointer of the code that made the memory access.
Address	The Address that was read / written to / executed and will be in the range of [beginAddress, endAddress) from the parameters to .Memory(). Note that the interval is half-open. That is, none of the returned events will have an address matching endAddress but there could be events matching endAddress – 1.

OBJECT	DESCRIPTION
Size	The size of the read/write/execute in bytes. This will typically be 8 bytes or less. In the event of code execution, it is the number of bytes in the instruction that was executed.
Value	The value that was read, written or executed. In the case of execution, it contains the code bytes for the instruction. Note the instruction bytes are listed in MSB order by the disassembler but will be stored in value in LSB order.

Remarks

The following access types are allowed in TTD.Memory queries:

- r - read
- w - write
- rw - read / write
- e - execute
- rwe - read / write / execute
- ec - execute /change

Note that this is a function that does computation, so it takes a while to run.

Example Usage

This example shows a grid display of all the positions in the trace where the four bytes of memory starting at 0x00a4fca0 were read access occurred. Click on any entry to drill down on each occurrence of memory access.

```
dx -g @$cursession.TTD.Memory(0x00a4fca0,0x00a4fca4, "r")
```

	(+) EventType	(+) ThreadId	(+) UniqueThreadId	(+) TimeStart	(+) TimeEnd	(+) AccessType	(+) IP	(+) Address	(+) Size	(+) Value
[0x0]	MemoryAccess	0x3168	0x2	15:16	15:16	Read	0x77a02196	0xa4fca0	0x4	0xa4fcf8
[0x1]	MemoryAccess	0x3168	0x2	15:18	15:18	Read	0x77a02198	0xa4fca0	0x4	0x77a26768
[0x2]	MemoryAccess	0x3168	0x2	15:36	15:36	Read	0x779c4305	0xa4fca0	0x4	0x779e0c21
[0x3]	MemoryAccess	0x3168	0x2	1E:26	1E:26	Read	0x779e0d07	0xa4fca0	0x4	0xa4fcf8
[0x4]	MemoryAccess	0x3168	0x2	2B:183	2B:183	Read	0x690b9c21	0xa4fca0	0x4	0x0
[0x5]	MemoryAccess	0x3168	0x2	3F:6	3F:6	Read	0x7562500a	0xa4fca0	0x4	0x1f000
[0x6]	MemoryAccess	0x3168	0x2	43:3	43:3	Read	0x779dfa9b	0xa4fca0	0x4	0x117a9480
[0x7]	MemoryAccess	0x3168	0x2	45:4	45:4	Read	0x779bc75b	0xa4fca0	0x4	0x799123f0
[0x8]	MemoryAccess	0x3168	0x2	47:3	47:3	Read	0x779df73b	0xa4fca0	0x4	0x117a9480
[0x9]	MemoryAccess	0x3168	0x2	47:A	47:A	Read	0x779df74b	0xa4fca0	0x4	0x117a9480
[0xa]	MemoryAccess	0x3168	0x2	4B:2F	4B:2F	Read	0xb0a688	0xa4fca0	0x4	0xa4fd00
[0xb]	MemoryAccess	0x3168	0x2	4B:118	4B:118	Read	0xb02e96	0xa4fca0	0x4	0x0
[0xc]	MemoryAccess	0x3168	0x2	4B:11C	4B:11C	Read	0xb02ea1	0xa4fca0	0x4	0x0
[0xd]	MemoryAccess	0x3168	0x2	4B:928	4B:928	Read	0x690c2bec	0xa4fca0	0x4	0x69191a60
[0xe]	MemoryAccess	0x3168	0x2	4B:939	4B:939	Read	0x690ca901	0xa4fca0	0x4	0x69108d19
[0xf]	MemoryAccess	0x3168	0x2	4B:941	4B:941	Read	0x779c3418	0xa4fca0	0x4	0x69191a80
[0x10]	MemoryAccess	0x3168	0x2	5D:113	5D:113	Read	0x690795e5	0xa4fca0	0x4	0x0
[0x11]	MemoryAccess	0x3168	0x2	5F:18	5F:18	Read	0x690795f0	0xa4fca0	0x4	0x0
[0x12]	MemoryAccess	0x3168	0x2	64:6D	64:6D	Read	0x690ca8fc	0xa4fca0	0x4	0x69191a60
[0x13]	MemoryAccess	0x3168	0x2	7A:CC	7A:CC	Read	0x77a340d9	0xa4fca0	0x4	0xf0
[0x14]	MemoryAccess	0x3168	0x2	B9:CC	B9:CC	Read	0x77a340d9	0xa4fca0	0x4	0xf0

You can click on the TimeStart fields in any of the events in the grid display, to display information for that event.

```
0:000> dx -r1 @$cursession.TTD.Memory(0x00a4fca0,0x00a4fca4, "r")[16].TimeStart
@$cursession.TTD.Memory(0x00a4fca0,0x00a4fca4, "r")[16].TimeStart                 : 5D:113 [Time Travel]
    Sequence         : 0x5d
    Steps            : 0x113
```

To move to the position in the trace that the event occurred, click on [Time Travel].

```
0:000> dx @$cursession.TTD.Memory(0x00a4fca0,0x00a4fca4, "r")[16].TimeStart.SeekTo()
@$cursession.TTD.Memory(0x00a4fca0,0x00a4fca4, "r")[16].TimeStart.SeekTo()
(27b8.3168): Break instruction exception - code 80000003 (first/second chance not available)
Time Travel Position: 5D:113

eax=0000004c ebx=00dd0000 ecx=00a4f89c edx=00a4f85c esi=00a4f89c edi=00b61046
eip=690795e5 esp=00a4f808 ebp=00a4f818 iopl=0         nv up ei pl nz na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000206
690795e5 ffb604040000    push    dword ptr [esi+404h] ds:002b:00a4fca0=00000000
```

In this example, all of the positions in the trace where the four bytes of memory starting at 0x1bf7d0 were read/write accessed are listed. Click on any entry to drill down on each occurrence of memory access.

```
0:000> dx @$cursession.TTD.Memory(0x1bf7d0,0x1bf7d4, "rw")
@$cursession.TTD.Memory(0x1bf7d0,0x1bf7d4, "rw")
    [0x0]
    [0x1]
    [0x2]
    [0x3]
     ...
```

In this example all of the postions in the trace where the four bytes of memory starting at 0x13a1710 were execute/change accessed are listed. Click on any occurrence to drill down on for addtional information on each occurrence of memory access.

```
0:000> dx -r1 @$cursession.TTD.Memory(0x13a1710,0x13a1714, "ec")[0]
@$cursession.TTD.Memory(0x13a1710,0x13a1714, "ec")[0]
    EventType        : MemoryAccess
    ThreadId         : 0x1278
    UniqueThreadId   : 0x2
    TimeStart        : 5B:4D [Time Travel]
    TimeEnd          : 5B:4D [Time Travel]
    AccessType       : Execute
    IP               : 0x13a1710
    Address          : 0x13a1710
    Size             : 0x1
    Value            : 0x55
```

See Also

Time Travel Debugging - Introduction to Time Travel Debugging objects

Time Travel Debugging - Overview

TTD Event Objects

10/25/2019 • 2 minutes to read • Edit Online

Description

TTD Event objects are used to give information about important events that happened during a time travel trace.

Properties

PROPERTY	DESCRIPTION
Type	Describes the type of event that happened. Possible values are: ThreadCreated, ThreadTerminated, ModuleLoaded, ModuleUnloaded, Exception

Children

OBJECT	DESCRIPTION
Position	A position object that describes the position the event occurred.
Module*	A module object containing information about the module that was loaded or unloaded.
Thread*	A thread object containing information about the thread that was created or terminated.
Exception*	An exception object containing information about the exception that was hit.

* - Existence of these child objects depends on the type of event

Example Usage

```
0:000> dx -r2 @$curprocess.TTD.Events.Where(t => t.Type == "Exception").Select(e => e.Exception)
@$curprocess.TTD.Events.Where(t => t.Type == "Exception").Select(e => e.Exception)
    [0x0]            : Exception of type CPlusPlus at PC: 0X777663B0
        Position         : 13B7:0 [Time Travel]
        Type             : CPlusPlus
        ProgramCounter   : 0x777663b0
        Code             : 0xe06d7363
        Flags            : 0x1
        RecordAddress    : 0x0
    [0x1]            : Exception of type Hardware at PC: 0XF1260D0
        Position         : BC0F:0 [Time Travel]
        Type             : Hardware
        ProgramCounter   : 0xf1260d0
        Code             : 0x80000003
        Flags            : 0x0
        RecordAddress    : 0x0
```

See Also

Time Travel Debugging - Introduction to Time Travel Debugging objects

Time Travel Debugging - Overview

TTD Exception Objects

10/25/2019 • 2 minutes to read • Edit Online

Description

TTD Exception objects are used to give information about exceptions that happened during a trace session.

Properties

PROPERTY	DESCRIPTION
Type	Describes the type of exception. Possible values are "Software" and "Hardware".
ProgramCounter	The instruction where the exception was thrown.
Code	The code of the exception.
Flags	The exception flags.
RecordAddress	Where in memory you can find the record of the exception.

Children

OBJECT	DESCRIPTION
Position	A position object that describes the position the exception occurred.

Example Usage

Information pending

See Also

Time Travel Debugging - Introduction to Time Travel Debugging objects

Time Travel Debugging - Overview

TTD Position Objects

10/25/2019 • 2 minutes to read • Edit Online

Description

Position objects are used to describe a position in a time travel trace. A position object is normally described by two hexadecimal numbers separated by a colon. The first of the hexadecimal numbers is the *Sequence* and the second is the *Steps*.

A position of FFFFFFFFFFFFFFFE:0 indicates the end of the trace.

Properties

PROPERTY	DESCRIPTION
Sequence	The sequencing point relevant to the position.
Steps	The number of steps from the sequence point in this thread to get to this position.

Methods

METHOD	DESCRIPTION
SeekTo()	Time travels to this position in the trace.

Example Usage

Information pending

See Also

Time Travel Debugging - Introduction to Time Travel Debugging objects

Time Travel Debugging - Overview

TTD Range Objects

10/25/2019 • 2 minutes to read • Edit Online

Description

TTD Range objects are used to give information about a range of time in a trace. These are generally used to describe the lifetime of a TTD thread object during a TTD session.

Children

OBJECT	DESCRIPTION
MinPosition	A position object that describes the earliest position relevant to the range.
MaxPosition	A position object that describes the latest position relevant to the range.

Example Usage

Information pending

See Also

Time Travel Debugging - Introduction to Time Travel Debugging objects

Time Travel Debugging - Overview

TTD Thread Objects

10/25/2019 • 2 minutes to read • Edit Online

Description

TTD Thread objects are used to give information about threads and their lifetime during a time travel trace.

Properties

PROPERTY	DESCRIPTION
UniqueId	A unique ID for the thread across the trace.
Id	The TID of the thread.

Children

OBJECT	DESCRIPTION
LifeTime	A TTD range object that describes the lifetime of the thread.
ActiveTime	A TTD range object that describes the time the thread was active.

Example Usage

Use this dx command to display all of the threads in the array.

```
0:0:000> dx -g @$curprocess.TTD.Threads
================================================================================================================
===
=                             = (+) UniqueId = (+) Id    = (+) Lifetime                = (+) ActiveTime
=
================================================================================================================
===
= [0x0] : UID: 2, TID: 0x2428 - 0x2          - 0x2428    - [0:0, 6F0C4:0]              - [50C63:0, 6F0C4:0]
=
= [0x1] : UID: 3, TID: 0x3520 - 0x3          - 0x3520    - [0:0, FFFFFFFFFFFFFFFE:0]    - [5115A:0, 56B07:0]
=
= [0x2] : UID: 4, TID: 0x18E8 - 0x4          - 0x18e8    - [0:0, FFFFFFFFFFFFFFFE:0]    - [52F65:0, 56B1E:0]
=
= [0x3] : UID: 5, TID: 0x5690 - 0x5          - 0x5690    - [0:0, FFFFFFFFFFFFFFFE:0]    - [5300D:0, 5D4FA:0]
=
= [0x4] : UID: 6, TID: 0x46FC - 0x6          - 0x46fc    - [0:0, FFFFFFFFFFFFFFFE:0]    - [53782:0, 5433B:0]
=
= [0x5] : UID: 7, TID: 0x58D0 - 0x7          - 0x58d0    - [0:0, FFFFFFFFFFFFFFFE:0]    - [542FE:0, 543B9:0]
=
= [0x6] : UID: 8, TID: 0x950  - 0x8          - 0x950     - [0:0, FFFFFFFFFFFFFFFE:0]    - [543C4:0, 544B8:0]
=
= [0x7] : UID: 9, TID: 0x4514 - 0x9          - 0x4514    - [0:0, 6D61B:0]              - [5DBBD:0, 6D61B:0]
=
================================================================================================================
===
```

Use this dx command to display information about the first thread in the array.

```
0:0:000 dx -r2 @$curprocess.TTD.Threads[0]
@$curprocess.TTD.Threads[0]                 : UID: 2, TID: 0x2428
    UniqueId         : 0x2
    Id               : 0x2428
    Lifetime         : [0:0, 6F0C4:0]
        MinPosition      : Min Position [Time Travel]
        MaxPosition      : 6F0C4:0 [Time Travel]
    ActiveTime       : [50C63:0, 6F0C4:0]
        MinPosition      : 50C63:0 [Time Travel]
        MaxPosition      : 6F0C4:0 [Time Travel]
```

The [Time Travel] links provide a link to SeekTo() the specific postion in the trace when the thread was active.

```
0:0:000> dx @$curprocess.TTD.Threads[0].@"ActiveTime".@"MinPosition".SeekTo()
Setting position: 50C63:0
@$curprocess.TTD.Threads[0].@"ActiveTime".@"MinPosition".SeekTo()
(40b4.2428): Break instruction exception - code 80000003 (first/second chance not available)
Time Travel Position: 50C63:0
ntdll!NtTestAlert+0x14:
00007ffb`e3e289d4 c3              ret
```

See Also

Time Travel Debugging - Introduction to Time Travel Debugging objects

Time Travel Debugging - Overview

Time Travel Debugging - JavaScript Automation

10/28/2019 • 2 minutes to read • Edit Online

Content Pending

See Also

Time Travel Debugging - Overview

Debugging Resources

6/25/2019 • 2 minutes to read • Edit Online

Use Debugging Tools for Windows to debug drivers, applications, and services on Windows systems. The core debugging engine in the tool set is called the Windows debugger. Starting with Windows 8, you can use Visual Studio as your interface to the Windows debugger. You can also use the traditional interfaces (WinDbg, CDB, and NTSD), which are included in Debugging Tools for Windows.

Getting Started with Debugging Tools for Windows

- Windows Debugging

Installing Debugging Tools for Windows

- Download and Install Debugging Tools for Windows

Debugger How-Tos

- Advanced Driver Debugging: Part 1 [media file]
- Advanced Driver Debugging: Part 2 [media file]
- Avoiding debugger searches for unneeded symbols
- Debugging Kernel-Mode Driver Framework Drivers
- Debugging WDF Drivers
- How to Enable Verbose Debug Tracing in Various Drivers and Subsystems
- Debugging a Driver
- **BCDEdit /dbgsettings**
- Tools for Debugging Drivers (WDK Documentation)

Knowledge base articles for debugging

A number of Knowledge Base articles are available for debugging related topics. This page provides a few examples of Product Support links that are relevant for debugging.

- 2816225: Enabling Debug mode causes Windows to hang if no Debugger is connected
- 311503: Use the Microsoft Symbol Server to obtain debug symbol files
- Q248413: INFO: NDIS Kernel Debugger Extensions
- Q121366: INFO: PDB and DBG Files - What They Are and How They Work
- Q121543: Setting Up for Remote Debugging
- Q129845: Blue Screen Preparation Before Calling Microsoft
- Q128372: How to Remove Symbols from Device Drivers
- Q97858: CTRL+C Exception Handling Under WinDbg
- Q102351: Debugging Console Apps Using Redirection
- Q117559: INF: How to Correlate Spid, Kpid, and Thread Instance
- Q121434: Specifying the Debugger for Unhandled User Mode Exceptions
- Q133722: HOWTO: Debug Flat Thunks
- Q148220: PRB: Debugger Reports "WARNING: symbols checksum is wrong"
- Q137199: PRB: Debuggers Cannot Reliably Use Debug Register Breakpoints

- Q100957: PRB: Debugging an Application Driven by MS-TEST
- Q130667: PRB: F12 Causes Hard-Coded Breakpoint Exception When Debugging
- Q173260: PRB: Synchronization Failure When Debugging
- Q168609: How to Use Display Heap (DH.EXE) Utility
- Q103861: INFO: Choosing the Debugger That the System Will Spawn

Related topics

DebugView Monitoring Tool

Driver Developer Community Resources

Driver Signing Requirements for Windows

Support for Developer Kits and Tools

Debugging Tools for Windows: New for Windows 10

6/25/2019 • 2 minutes to read • Edit Online

Windbg Preview

For the lastest news on Windows Debugging tools, see WinDbg Preview - What's New.

Windows 10, version 1703

This section describes new debugging tools in Windows 10, version 1703.

- Eight new JavaScript topics including JavaScript Debugger Scripting
- Updates to the **dx (Display Debugger Object Model Expression)** command, to include new command capabilities.
- New **dtx (Display Type - Extended Debugger Object Model Information)** command.
- New **!ioctldecode** command.
- Updated Debugger Engine Reference to include additional interfaces and structures.
- Updates to Configuring tools.ini to document additional options for the command line debuggers.
- Published 75 previously undocumented stop codes in Bug Check Code Reference.
- New Supported Ethernet NICs for Network Kernel Debugging in Windows 10 topic.

Windows 10, version 1607

This section describes new debugging tools in Windows 10, version 1607.

- New topic about Debugging a UWP app using WinDbg.
- Updates to the 30 most-viewed developer bug check topics in Bug Check Code Reference.

Windows 10

- **.settings (Set Debug Settings)** - New command that allows you to set, modify, display, load and save settings in the Debugger.Settings namespace.
- **dx (Display NatVis Expression)** - Describes the new dx debugger command, which displays object information using the NatVis extension model and LINQ support.
- New commands that work with the NatVis visualization files in the debugger environment.
  - **.nvlist (NatVis List)**
  - **.nvload (NatVis Load)**
  - **.nvunload (NatVis Unload)**
  - **.nvunloadall (NatVis Unload All)**
- Bluetooth Extensions (Bthkd.dll)
- Storage Kernel Debugger Extensions
- New Symproxy information including SymProxy Automated Installation. In addition the following topics are updated to cover new SymProxy functionality:
  - HTTP Symbol Stores
  - SymProxy
  - Installing SymProxy
  - Configuring the Registry

    - Configuring IIS for SymProxy
- **CDB Command-Line Options** - Updated to include new command line options.
- **!analyze** - Updated to include information about using this extension with UMDF 2.15.
- **!wdfkd.wdfcrashdump**- Updated to include information about using this extension with UMDF 2.15
- **!irp** - Updated. Starting with Windows 10 the IRP major and minor code text is displayed in command output.
- Using Debugger Markup Language - Updated to describe new right click behavior available in the Debugger Markup Language (DML).
- Crash dump analysis using the Windows debuggers (WinDbg) - Performance has increased in taking a memory dump over KDNET.
- Debug Universal Drivers - Step by Step Lab (Echo Kernel-Mode)- New step by step lab that shows how to use WinDbg to debug the sample KMDF echo driver.

Looking to download the Debugging Tools?

For information on downloading the debugging tools, see Download Debugging Tools for Windows.

Tools Included in Debugging Tools for Windows

12/5/2018 • 2 minutes to read • Edit Online

Debugging Tools for Windows includes several tools in addition to the debugging engine and the Debugging Environments. The tools are in the installation directory of Debugging Tools for Windows.

ADPlus
Automatically create memory dump files and log files with debug output from one or more processes.

DumpChk
Validate a memory dump file.

GFlags
Control registry keys and other settings.

Kill
Terminate a process.

Logger and LogViewer
Record and display function calls and other actions of a program.

PLMDebug
Use the Windows debugger to debug Windows app, which run under Process Lifecycle Management (PLM). With PLMDebug, you can take manual control of suspending, resuming, and terminating a Windows app.

Remote Tool
Remotely control any console program, including KD, CDB, and NTSD. See Remote Debugging Through Remote.exe.

TList
List all running processes.

UMDH
Analyze heap allocations.

USBView
Display USB host controllers and connected devices.

DbgRpc (Dbgrpc.exe)
Display Microsoft Remote Procedure Call (RPC) state information. See RPC Debugging and Using the DbgRpc Tool.

KDbgCtrl (Kernel Debugging Control, Kdbgctrl.exe)
Control and configure the kernel debugging connection. See Using KDbgCtrl.

SrcSrv
A source server that can be used to deliver source files while debugging.

SymSrv
A symbol server that the debugger can use to connect to a symbol store.

SymProxy
Create a single HTTP symbol server on your network that all your debuggers can point to. This has the benefit of pointing to multiple symbol servers (both internal and external) with a single symbol path, handling all authentication, and increasing performance via symbol caching. Symproxy.dll is in the SymProxy folder in the

installation directory.

SymChk
Compare executable files to symbol files to verify that the correct symbols are available.

SymStore
Create a symbol store. See Using SymStore.

AgeStore
Removes old entries in the downstream store of a symbol server or a source server.

DBH
Display information about the contents of a symbol file.

PDBCopy
Remove private symbol information from a symbol file, and control which public symbols are included in the file.

DbgSrv
A process server used for remote debugging. See Process Servers (User Mode).

KdSrv
A KD connection server used for remote debugging.See KD Connection Servers (Kernel Mode).

DbEngPrx
A repeater (small proxy server) used for remote debugging. See Repeaters.

Breakin (Breakin.exe)
Causes a user-mode break to occur in a process. For help, open a Command Prompt window, navigate to the installation directory, and enter **breakin /?**.

List (File List Utility) (List.exe)
For help, open a Command Prompt window, navigate to the installation directory, and enter **list /?**.

RTList (Remote Task List Viewer) (Rtlist.exe)
List running processes via a DbgSrv process server. For help, open a Command Prompt window, navigate to the installation directory, and enter **rtlist /?**.

Installation Directory

The default installation directory for 64 bit OS installs for the debugging tools is C:\Program Files (x86)\Windows Kits\10\Debuggers\. If you have a 32-bit OS, you can find the Windows Kits folder under C:\Program Files. To determine if you should use the 32 bit or 64 bit tools, see Choosing the 32-Bit or 64-Bit Debugging Tools.

Related topics

Tools Related to Debugging Tools for Windows

ADPlus

12/5/2018 • 2 minutes to read • Edit Online

ADPlus V7.0 is a total rewrite of ADPlus. ADPlus V7.0 is written in managed code, which allows us to easily add new features. ADPlus.exe keeps the basic functionality of ADPlus.vbs and adds some additional features. Also, there is a new companion called ADPlusManager, which extends ADPlus to a distributed environment like an HPC computer cluster.

Where to get ADPlus

ADPlus is included in Debugging Tools for Windows.

For ADPlus V7.0 documentation, see adplus.doc in the installation folder for Debugging Tools for Windows.

Related topics

Tools Included in Debugging Tools for Windows

DumpChk

12/5/2018 • 4 minutes to read • Edit Online

DumpChk (the Microsoft Crash Dump File Checker tool) is a program that performs a quick analysis of a crash dump file. This enables you to see summary information about what the dump file contains. If the dump file is corrupt in such a way that it cannot be opened by a debugger, DumpChk reveals this fact.

Where to get DumpChk

DumpChk.exe is included in Debugging Tools for Windows.

DumpChk command-line options

```
DumpChk [-y SymbolPath] DumpFile
```

Parameters

-y *SymbolPath*
SymbolPath specifies where DumpChk is to search for symbols. Symbol information may be necessary for some dump files. It can also help to improve the information shown in the dump file by allowing symbol names to be resolved.

DumpFile
DumpFile specifies the crash dump file that is to be analyzed. This may include an absolute or relative directory path or universal naming convention (UNC) path. If *DumpFile* contains spaces, it must be enclosed in quotation marks.

Using DumpChk

Here is an example in which the dump file is corrupt. The error shown at the end, `DebugClient cannot open DumpFile`, indicates that some kind of corruption must have occurred:

```
C:\Debuggers> dumpchk c:\mydir\dumpfile2.dmp

Loading dump file c:\mydir\dumpfile2.dmp

Microsoft (R) Windows Debugger Version 6.9.0003.113 X86
Copyright (C) Microsoft. All rights reserved.


Loading Dump File [c:\mydir\dumpfile2.dmp]
Could not match Dump File signature - invalid file format
Could not open dump file [c:\mydir\dumpfile2.dmp], HRESULT 0x80004002
    "No such interface supported"
**** DebugClient cannot open DumpFile - error 80004002
```

Because this display does not end with the words `Finished dump check`, the dump file is corrupt. The error message at the end explains that the dump file could not be opened.

Note that other errors may be listed, some of which are actually benign. For example, the following error message does not represent a problem:

```
error 3 InitTypeRead( nt!_PEB at 7ffd5000)
```

Here is an example of DumpChk run on a healthy user-mode minidump. The display begins with an overall summary of the dump file, and then gives detailed information about what data is contained in the dump file:

```
C:\Debuggers> dumpchk c:\mydir\dumpfile1.dmp

Loading dump file c:\mydir\dumpfile1.dmp

Microsoft (R) Windows Debugger Version 6.9.0003.113 X86
Copyright (C) Microsoft. All rights reserved.


Loading Dump File [c:\mydir\dumpfile1.dmp]
User Mini Dump File with Full Memory: Only application data is available

Symbol search path is: srv*C:\CODE\LocalStore*\\symbols\symbols
Executable search path is:
Windows Vista Version 6000 MP (2 procs) Free x86 compatible
Product: WinNt, suite: SingleUserTS
Debug session time: Tue Jun 17 02:28:23.000 2008 (GMT-7)
System Uptime: 0 days 15:43:52.861
Process Uptime: 0 days 0:00:26.000
...
This dump file has an exception of interest stored in it.
The stored exception information can be accessed via .ecxr.

----- User Mini Dump Analysis

MINIDUMP_HEADER:
Version         A793 (6903)
NumberOfStreams 12
Flags           1826
                0002 MiniDumpWithFullMemory
                0004 MiniDumpWithHandleData
                0020 MiniDumpWithUnloadedModules
                0800 MiniDumpWithFullMemoryInfo
                1000 MiniDumpWithThreadInfo

Streams:
Stream 0: type ThreadListStream (3), size 00000064, RVA 000001BC
  2 threads
  RVA 000001C0, ID 1738, Teb:000000007FFDF000
  RVA 000001F0, ID 1340, Teb:000000007FFDE000
Stream 1: type ThreadInfoListStream (17), size 0000008C, RVA 00000220
  RVA 0000022C, ID 1738
  RVA 0000026C, ID 1340
Stream 2: type ModuleListStream (4), size 00000148, RVA 000002AC
  3 modules
  RVA 000002B0, 00400000 - 00438000: 'C:\CODE\TimeTest\Debug\TimeTest.exe'
  RVA 0000031C, 779c0000 - 77ade000: 'C:\Windows\System32\ntdll.dll'
  RVA 00000388, 76830000 - 76908000: 'C:\Windows\System32\kernel32.dll'
Stream 3: type Memory64ListStream (9), size 00000290, RVA 00001D89
  40 memory ranges
  RVA 0x2019 BaseRva
  range#    RVA      Address      Size
       0 00002019    00010000   00010000
       1 00012019    00020000   00005000
       2 00017019    0012e000   00002000

 (additional stream data deleted)

Stream 9: type UnusedStream (0), size 00000000, RVA 00000000
Stream 10: type UnusedStream (0), size 00000000, RVA 00000000
Stream 11: type UnusedStream (0), size 00000000, RVA 00000000
```

```
Windows Vista Version 6000 MP (2 procs) Free x86 compatible
Product: WinNt, suite: SingleUserTS
kernel32.dll version: 6.0.6000.16386 (vista_rtm.061101-2205)
Debug session time: Tue Jun 17 02:28:23.000 2008 (GMT-7)
System Uptime: 0 days 15:43:52.861
Process Uptime: 0 days 0:00:26.000
  Kernel time: 0 days 0:00:00.000
  User time: 0 days 0:00:00.000
PEB at 7ffd9000
    InheritedAddressSpace:    No
    ReadImageFileExecOptions: No
    BeingDebugged:            Yes
    ImageBaseAddress:         00400000
    Ldr                       77a85d00
    Ldr.Initialized:          Yes
    Ldr.InInitializationOrderModuleList: 002c1e30 . 002c2148
    Ldr.InLoadOrderModuleList:           002c1da0 . 002c2138
    Ldr.InMemoryOrderModuleList:         002c1da8 . 002c2140
            Base TimeStamp                     Module
          400000 47959d85 Jan 21 23:38:45 2008 C:\CODE\TimeTest\Debug\TimeTest.exe
        779c0000 4549bdc9 Nov 02 02:43:37 2006 C:\Windows\system32\ntdll.dll
        76830000 4549bd80 Nov 02 02:42:24 2006 C:\Windows\system32\kernel32.dll
    SubSystemData:     00000000
    ProcessHeap:       002c0000
    ProcessParameters: 002c14c0
    WindowTitle:  'C:\CODE\TimeTest\Debug\TimeTest.exe'
    ImageFile:    'C:\CODE\TimeTest\Debug\TimeTest.exe'
    CommandLine:  '\CODE\TimeTest\Debug\TimeTest.exe'
    DllPath:      'C:\CODE\TimeTest\Debug;C:\Windows\system32;C:\Windows\system;
    Environment:  002c0808
        =C:=C:\CODE
        =ExitCode=00000000
        ALLUSERSPROFILE=C:\ProgramData
        AVENGINE=C:\PROGRA~1\CA\SHARED~1\SCANEN~1
        CommonProgramFiles=C:\Program Files\Common Files
        COMPUTERNAME=EMNET
        ComSpec=C:\Windows\system32\cmd.exe
        configsetroot=C:\Windows\ConfigSetRoot
        FP_NO_HOST_CHECK=NO
        HOMEDRIVE=C:
        NUMBER_OF_PROCESSORS=2
        OS=Windows_NT
        Path=C:\DTFW\200804~2.113\winext\arcade;C:\Windows\system32
        PATHEXT=.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC
        PROCESSOR_ARCHITECTURE=x86
        PROCESSOR_IDENTIFIER=x86 Family 6 Model 15 Stepping 13, GenuineIntel
        PROCESSOR_LEVEL=6
        PROCESSOR_REVISION=0f0d
        ProgramData=C:\ProgramData
        ProgramFiles=C:\Program Files
        PROMPT=$P$G
        PUBLIC=C:\Users\Public
        RoxioCentral=C:\Program Files\Common Files\Roxio Shared\9.0\Roxio Central33\
        SESSIONNAME=Console
        SystemDrive=C:
        SystemRoot=C:\Windows
        USERDNSDOMAIN=NORTHSIDE.COMPANY.COM
        USERDOMAIN=NORTHSIDE
        USERNAME=myname
        USERPROFILE=C:\Users\myname
        WINDBG_DIR=C:\DTFW\200804~2.113
        windir=C:\Windows
        WINLAYTEST=200804~2.113
        _NT_SOURCE_PATH=C:\mysources
        _NT_SYMBOL_PATH=C:\mysymbols
Finished dump check
```

The output begins by identifying the characteristics of the dump file - in this case, a user-mode minidump with full memory information, including application data but not operating-system data. This is followed by the symbol path being used by DumpChk, and then a summary of the dump file contents.

Because this display ends with the words `Finished dump check`, the dump file is probably not corrupt, and can be opened by a debugger. However, more subtle forms of corrruption might still be present in the file.

Related topics

Tools Included in Debugging Tools for Windows

GFlags

12/5/2018 • 2 minutes to read • Edit Online

GFlags (the Global Flags Editor), gflags.exe, enables and disables advanced debugging, diagnostic, and troubleshooting features. It is most often used to turn on indicators that other tools track, count, and log.

Where to get GFlags

GFlags is included in the Debugging Tools for Windows 10 (WinDbg).

After the debugging tools are installed, gflags.exe for use on 64 bit Windows is installed by default to the following directory.

```
C:\Program Files (x86)\Windows Kits\10\Debuggers\x64
```

If you are running a 32 bit version of Windows, use the 32 bit version of gflags.exe located here.

```
C:\Program Files (x86)\Windows Kits\10\Debuggers\x86
```

Overview of GFlags

Driver developers and testers often use GFlags to turn on debugging, logging and test features, either directly, or by including GFlags commands in a test script. The page heap verification features can help you to identify memory leaks and buffer errors in kernel-mode drivers.

GFlags has both a dialog box and a command-line interface. Most features are available from both interfaces, but some features are accessible from only one of the interfaces. (See GFlags Details.)

Features

- Page heap verification. GFlags now includes the functions of PageHeap (pageheap.exe), a tool that enables heap allocation monitoring. PageHeap was included in previous versions of Windows.
- No reboot required for the Special Pool feature. On Windows Vista and later versions of Windows, you can enable, disable, and configure the Special Pool feature without restarting ("rebooting") the computer. For information, see Special Pool.
- Object Reference Tracing. A new flag enables tracing of object referencing and object dereferencing in the kernel. This new feature of Windows detects when an object reference count is decremented too many times or not decremented even though an object is no longer used. This flag is supported only in Windows Vista and later versions of Windows.
- New dialog box design. The GFlags dialog box has tabbed pages for easier navigation.

Requirements

To use most GFlags features, including setting flags in the registry or in kernel mode, or enabling page heap verification, you must be a member of the Administrators group on the computer. However, prior to Windows Vista, users with at least Guest account access can launch a program from the **Global Flags** dialog box.

When features do not work, or work differently, on particular operating system versions, the differences are explained in the description of the feature.

This topic includes:

GFlags Overview

GFlags Details

GFlags Commands

GFlags Flag Table

GFlags and PageHeap

Global Flags Dialog Box

GFlags Examples

Global Flag Reference

Note Incorrect use of this tool can degrade system performance or prevent Windows from starting, requiring you to reinstall Windows.

Important Pool tagging is permanently enabled on Windows Server 2003 and later versions of Windows, including Windows Vista. On these systems, the **Enable pool tagging** check box on the **Global Flags** dialog box is dimmed and commands to enable or disable pool tagging fail.

Related topics

Tools Included in Debugging Tools for Windows

GFlags Overview

12/5/2018 • 2 minutes to read • Edit Online

GFlags (gflags.exe), the Global Flags Editor, enables and disables advanced internal system diagnostic and troubleshooting features. You can run GFlags from a Command Prompt window or use its graphical user interface dialog box.

For information on how to install and locate gflags.exe, see GFlags.

Use GFlags to activate the following features:

Registry
Set system-wide debugging features for all processes running on the computer. These settings are stored in the **GlobalFlag** registry entry (**HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\GlobalFlag**). They take effect when you restart Windows and remain effective until you change them and restart again.

Kernel flag settings
Set debugging features for this session. These settings are effective immediately, but are lost when Windows shuts down. The settings affect all processes started after this command completes.

Image file settings
Set debugging features for a particular program. These settings are stored in a GlobalFlag registry entry for each program (**HKEY_LOCAL_MACHINE\ SOFTWARE\ Microsoft\ Windows NT\ CurrentVersion\ Image File Execution Options\ *ImageFileName*\ GlobalFlag**). They take effect when you restart the program and remain effective until you change them.

Debugger
Specify that a particular program always runs in a debugger. This setting is stored in the registry. It is effective immediately and remains effective until you change it. (This feature is available only in the **Global Flags** dialog box.)

Launch
Run a program with the specified debugging settings. The debugging settings are effective until the program stops. (This feature is available only from the **Global Flags** dialog box.)

Special Pool
Request that allocation with a specified pool tag or of a specified size are filled from the special pool. This feature helps you to detect and identify the source of errors in kernel pool use, such as writing beyond the allocated memory space, or referring to memory that has already been freed.

Beginning in Windows Vista, you can enable, disable, and configure the special pool feature (**Kernel Special Pool Tag**) as a kernel flags setting, which does not require a reboot, or as a registry setting, which requires a reboot.

Page heap verification
Enable, disable, and configure page heap verification for a program. When enabled, page heap monitors dynamic heap memory operations, including allocation and free operations, and causes a debugger break when it detects a heap error.

Silent process exit
Enable, disable, and configure monitoring and reporting of silent exits for a process. You can specify actions that occur when a process exits silently, including notification, event logging, and creation of dump files. For more information, see Monitoring Silent Process Exit.

GFlags Details

7/15/2019 • 2 minutes to read • Edit Online

GFlags enables and disables system features by editing the Windows registry and internal settings. This section explains the operation of GFlags in detail and includes tips for using GFlags most efficiently.

General Information

- To display the GFlags dialog box, at the command line, type **gflags** (with no parameters).
- GFlags system-level registry settings appear in the registry immediately, but do not take effect until you restart the system.
- GFlags image file registry settings appear in the registry immediately, but do not take effect until you restart the process.
- The debugger and launch features in the GFlags dialog box are program specific. You can only set them on one image file at a time.

Flag Details

- To clear all flags, set the flag to -FFFFFFFF. Setting the flag to 0 adds 0 to the current flag value.
- When you set the flags for an image file to FFFFFFFF (0xFFFFFFFF), Windows clears all flags for the image file and deletes the **GlobalFlag** entry in the image file registry key. The image file registry key is retained.

Dialog Box and Command Line

You can run GFlags by using its handy dialog box or from the command line. Most features are available in both forms, with the following exceptions.

Dialog box only

- Launch. Start a program using the specified flags.
- Run the program in a debugger.
- Special Pool on systems prior to Windows Vista. On Windows Vista and later versions of Windows, you can configure the Special Pool feature at the command line or in the Gflags dialog box.

Command line only

- Set the size of the user mode stack trace database (/tracedb).
- Set page heap verification options.

Registry Information

GFlags settings that are saved between sessions are stored in the registry. You can use the registry APIs, Regedit, or reg.exe to query or change these values. The following table lists the types of settings and where they are stored in the registry.

TYPE OF SETTING	REGISTRY LOCATION
Systemwide settings ("Registry")	HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager**GlobalFlag**

TYPE OF SETTING	REGISTRY LOCATION
Program-specific settings ("Image file") for all users of the computer.	HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options*ImageFileName***GlobalFlag**
Silent exit settings for a specific program ("Silent Process Exit") for all users of the computer.	HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\SilentProcessExit***ImageFileName***
Page heap options for an image file for all users of the computer	HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options*ImageFileName***PageHeapFlags**
User mode stack trace database size (**tracedb**)	HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options*ImageFileName***StackTraceDatabaseSizeInMb**
Create user mode stack trace database (ust, 0x1000) for an image file	Windows adds the image file name to the value of the USTEnabled registry entry (HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options**USTEnabled**).
Load image using large pages if possible	HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options*ImageFileName***UseLargePages**.
Special Pool (Kernel Special Pool Tag)	HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management**PoolTag**
Verify Start / Verify End	HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\PoolTagOverruns. The **Verify Start** option sets the value to 0. The **Verify End** option sets the value to 1.
Debugger for an image file	HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options*ImageFileName***Debugger**
Object Reference Tracing	HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Kernel**ObTraceProcessName**, **ObTracePermanent** and **ObTracePoolTags**

GFlags Command Overview

10/21/2019 • 14 minutes to read • Edit Online

For general information on how to install and locate gflags.exe, see GFlags.

You can use the GFlags commands and the Global Flags Dialog Box interchangeably.

GFlags Command Usage

To use GFlags, type the following commands at the command line.

To open the GFlags dialog box:

```
gflags
```

To set or clear global flags in the registry:

```
gflags /r [{+ | -}Flag [{+ | -}Flag...]]
```

To set or clear global flags for the current session:

```
gflags /k [{+ | -}Flag [{+ | -}Flag...]]
```

To set or clear global flags for an image file:

```
gflags /i ImageFile [{+ | -}Flag [{+ | -}Flag...]]
gflags /i ImageFile /tracedb SizeInMB
```

To set or clear the Special Pool feature (Windows Vista and later)

```
gflags {/r | /k} {+ | -}spp {PoolTag | 0xSize}
```

To enable or disable the Object Reference Tracing feature (Windows Vista and later)

```
gflags {/ro | /ko} [/p] [/i ImageFile | /t PoolTag;[PoolTag...]]
```

```
gflags {/ro | /ko} /d
```

To enable and configure page heap verification:

```
gflags /p /enable ImageFile  [ /full [/backwards] | /random Probability | /size SizeStart SizeEnd | /address
AddressStart AddressEnd | /dlls DLL [DLL...] ]
[/debug ["DebuggerCommand"] | /kdebug] [/unaligned] [/notraces] [/fault Rate [TimeOut]] [/leaks] [/protect]
[/no_sync] [/no_lock_checks]
```

To disable page heap verification:

```
gflags /p [/disable ImageFile] [/?]
```

To display help:

```
gflags /?
```

Parameters

Flag
Specifies a three-letter abbreviation (*FlagAbbr*) or hexadecimal value (*FlagHex*) that represents a debugging feature. The abbreviations and hexadecimal values are listed in the GFlags Flag Table.

Use one of the following flag formats:

FORMAT	DESCRIPTION
{**+** | **-**}*FlagAbbr*	Sets (+) or clears (-) the flag represented by the flag abbreviation. Without a plus (+) or minus (-) symbol, the command has no effect.
[**+** | **-**]*FlagHex*	Adds (+) or subtracts (-) the hexadecimal value of a flag. A flag is set when its value is included in the sum. Add (+) is the default. Enter a hexadecimal value (without 0x) that represents a single flag or enter the sum of hexadecimal values for multiple flags.

ImageFile
Specifies the name of an executable file, including the file name extension (for example, notepad.exe or mydll.dll).

/r
Registry. Displays or changes system-wide debugging flags stored in the registry. These settings take effect when you restart Windows and remain effective until you change them.

With no additional parameters, **gflags /r** displays the system-wide flags set in the registry.

/k
Kernel flag settings. Displays or changes system-wide debugging flags for this session. These settings are effective immediately, but are lost when Windows shuts down. The settings affect processes started after this command completes.

With no additional parameters, **gflags /k** displays system-wide flags set for the current session.

/i
Image file settings. Displays or changes debugging flags for a particular process. These settings are stored in the registry. They are effective for all new instances of this process and they remain effective until you change them.

With no additional parameters, **gflags /i** *ImageFile* displays the flags set for the specified process.

/tracedb *SizeInMB*
Sets the maximum size of the user-mode stack trace database for the process. To use this parameter, the Create user mode stack trace database (ust) flag must be set for the process.

SizeInMB is a whole number representing the number of megabytes in decimal units. The default value is the minimum size, 8 MB; there is no maximum size. To revert to the default size, set *SizeInMB* to 0.

spp
(Windows Vista and later.) Sets or clears the Special Pool feature. For an example, see Example 14: Configuring Special Pool.

PoolTag
(Windows Vista and later.) Specifies a pool tag for the Special Pool feature. Use only with the **spp** flag.

Enter a four-character pattern for *PoolTag*, such as Tag1. It can include the **?** (substitute for any single character) and ***** (substitute for multiple characters) wildcard characters. For example, Fat* or Av?4. Pool tags are always case-sensitive.

0x*Size*
(Windows Vista and later.) Specifies a size range for the Special Pool feature. Use only with the **spp** flag. For guidance on selecting a size value, see "Selecting an Allocation Size" in Special Pool.

/ro
Enables, disables, and displays Object Reference Tracing settings in the registry. To make a change to this setting effective, you must restart Windows.

Without additional parameters, **/ro** displays the Object Reference Tracing settings in the registry.

To enable Object Reference Tracing, you must include at least one pool tag (**/t** *PoolTag*) or one image file (**/i** ImageFile) in the command. For details, see Example 15: Using Object Reference Tracing.

The following table lists the subparameters that are valid with **/ro**.

/t *PoolTags*	Limits the trace to objects with the specified pool tags. Use a semicolon (**;**) to separate tag names. Enter up to 16 pool tags. Enter a four-character pattern for *PoolTags*, such as Tag1. If you specify more than one pool tag, Windows traces objects with any of the specified pool tags . If you do not specify any pool tags, Windows traces all objects that are created by the image.
/i *ImageFile*	Limits the trace to objects that are created by processes with the specified image file. You can specify only one image file with the **/i** parameter. Enter an image file name, such as notepad.exe, with up to 64 characters. "System" and "Idle" are not valid image names. If you do not specify an image file, Windows traces all objects with the specified pool tags. If you specify both an image file (**/i**) and one or more pool tags (**/t**), Windows traces objects with any of the specified pool tags that are created by the specified image.
/d	Clears the Object Reference Tracing feature settings. When used with **/ro**, it clears the settings in the registry.
/p	Permanent. The trace data is retained until Object Reference Tracing is disabled, or the computer is shut down or restarted. By default, the trace data for an object is discarded when the object is destroyed.

/ko
Enables, disables, and displays kernel flag (run time) Object Reference Tracing settings. Changes to this setting are effective immediately, but are lost when the system is shut down or restarted. For details, see Example 15: Using Object Reference Tracing.

Without additional parameters, **/ko** displays the kernel flag (run time) Object Reference Tracing settings.

To enable Object Reference Tracing, you must include at least one pool tag (**/t** *PoolTag*) or one image file (**/i** *ImageFile*) in the command.

The following table lists the subparameters that are valid with **/ko**.

/t *PoolTags*	Limits the trace to objects with the specified pool tags. Use a semicolon (**;**) to separate tag names. Enter up to 16 pool tags. Enter a four-character pattern for *PoolTags*, such as Tag1. If you specify more than one pool tag, Windows traces objects with any of the specified pool tags. If you do not specify any pool tags, Windows traces all objects that are created by the image.
/i *ImageFile*	Limits the trace to objects that are created by processes with the specified image file. You can specify only one image file with the **/i** parameter. If you do not specify an image file, Windows traces all objects with the specified pool tags. If you specify both an image file (**/i**) and one or more pool tags (**/t**), Windows traces objects with any of the specified pool tags that are created by the specified image.
/d	Clears the Object Reference Tracing feature settings. When used with **/ro**, it clears the settings in the registry.
/p	Permanent. The trace data is retained until Object Reference Tracing is disabled, or the computer is shut down or restarted. By default, the trace data for an object is discarded when the object is destroyed.

/p
Sets page heap verification options for a process.

With no additional parameters, **gflags /p** displays a list of image files for which page heap verification is enabled.

Page heap verification monitors dynamic heap memory operations, including allocate operations and free operations, and causes a debugger break when it detects a heap error.

/disable *ImageFile*
Turns off page heap verification (standard or full) for the specified image file.

This command is equivalent to turning off the Enable page heap (hpa) flag for a process (**gflags /i** *ImageFile* **-hpa**). You can use the commands interchangeably.

/enable *ImageFile*
Turns on page heap verification for the specified image file.

By default, the **/enable** parameter turns on *standard* page heap verification for the image file. To enable *full* page heap verification for the image file, add the **/full** parameter to the command or use the **/i** parameter with the **+hpa** flag.

/full
Turns on full page heap verification for the process. Full page heap verification places a zone of reserved virtual memory at the end of each allocation.

Using this parameter is equivalent to turning on the Enable page heap (hpa) flag for a process (**gflags /i** *ImageFile* **+hpa**). You can use the commands interchangeably.

/backwards
Places the zone of reserved virtual memory at the beginning of an allocation, rather than at the end. As a result, the debugger traps overruns at the beginning of the buffer, instead of those at the end of the buffer. Valid only with the **/full** parameter.

/random *Probability*
Chooses full or standard page heap verification for each allocation, based on the specified probability.

Probability is a decimal integer from 0 through 100 representing the probability of full page heap verification. A probability of 100 is the same as using the **/full** parameter. A probability of 0 is the same as using standard page heap verification.

/size *SizeStart SizeEnd*
Enables full page heap verification for allocations within the specified size range and enables standard page heap verification for all other allocations by the process.

SizeStart and *SizeEnd* are decimal integers. The default is standard page heap verification for all allocations.

/address *AddressStart AddressEnd*
Enables full page heap verification for memory allocated while a return address in the specified address range is on the run-time call stack. It enables standard page heap verification for all other allocations by the process.

To use this feature, specify a range that includes the addresses of all functions that call the function with the suspect allocation. The address of the calling function will be on the call stack when the suspect allocation occurs.

AddressStart and *AddressEnd* specify the address range searched in allocation stack traces. The addresses are specified in hexadecimal format, such as, 0xAABBCCDD.

On Windows Server 2003 and earlier systems, the **/address** parameter is valid only on *x*86-based computers. On Windows Vista,: it is valid on all supported architectures.

/dlls *DLL*[**,** *DLL*...]
Enables full page heap verification for allocations requested by the specified DLLs and standard page heap verification for all other allocations by the process.

DLL is the name of a binary file, including its file name extension. The specified file must be a function library that the process loads during execution.

/debug
Automatically launches the process specified by the **/enable** parameter under a debugger.

By default, this parameter uses the NTSD debugger with the command line **ntsd -g -G -x** and with page heap enabled, but you can use the *DebuggerCommand* variable to specify a different debugger and command line.

For information about NTSD, see Debugging Using CDB and NTSD.

This option is useful for programs that are difficult to start from a command prompt and those that are started by other processes.

"*DebuggerCommand***"**
Specifies a debugger and the command sent to the debugger. This quoted string can include a fully qualified path to the debugger, the debugger name, and command parameters that the debugger interprets. The quotation marks are required.

If the command includes a path to the debugger, the path cannot contain any other quotation marks. If other quotation marks appear, the command shell (*cmd.exe*) will misinterpret the command.

/kdebug
Automatically launches the process specified by the **/enable** parameter under the NTSD debugger with the command line **ntsd -g -G -x**, with page heap enabled, and with control of NTSD redirected to the kernel debugger.

For information about NTSD, see Debugging Using CDB and NTSD.

/unaligned
Aligns the end of each allocation at an end-of-page boundary, even if doing so means that the starting address is not aligned on an 8-byte block. By default, the heap manager guarantees that the starting address of an allocation is aligned on an 8-byte block.

This option is used to detect off-by-one-byte errors. When this parameter is used with the **/full** parameter, the zone of reserved virtual memory begins just after the last byte of the allocation and an immediate fault occurs when a process tries to read or write even one byte beyond the allocation.

/decommit
This option is no longer valid. It is accepted, but ignored.

The PageHeap program (pageheap.exe) included in Windows 2000 implemented full page heap verification by placing an inaccessible page after an allocation. In that tool, the **/decommit** parameter substituted a zone of reserved virtual memory for the inaccessible page. In this version of GFlags, a zone of reserved virtual memory is always used to implement full page heap verification.

/notraces
Specifies that run-time stack traces are not saved.

This option improves performance slightly, but it makes debugging much more difficult. This parameter is valid, but its use is not recommended.

/fault
Forces the program's memory allocations to fail at the specified rate and after the specified time-out.

This parameter inserts heap allocation errors into the image file being tested (a practice known as "fault injection") so that some memory allocations fail, as might occur when the program runs in low memory conditions. This test helps to detect errors in handling allocation failure, such as failing to release resources.

Rate	Specifies a decimal integer from 1 (.01%) through 10000 (100%) representing the probability that an allocation will fail. The default is 100 (1%).
TimeOut	Determines the time interval between the start of the program and the start of the fault injection routines. *TimeOut* is measured in seconds. The default is 5 (seconds).

/leaks
Checks for heap leaks when a process ends.

The **/leaks** parameter disables full page heap. When **/leaks** is used, the **/full** parameter and parameters that modify the **/full** parameter, such as **/backwards**, are ignored, and GFlags performs standard page heap verification with a leak check.

/protect
Protects heap internal structures. This test is used to detect random heap corruptions. It can make execution significantly slower.

/no_sync
Checks for unsynchronized access. This parameter causes a break if it detects that a heap created with the HEAP_NO_SERIALIZE flag is accessed by different threads.

Do not use this flag to debug a program that includes a customized heap manager. Functions that synchronize heap access cause the page heap verifier to report synchronization faults that do not exist.

/no_lock_checks
Disables the critical section verifier.

/?
Displays help for GFlags. With **/p**, **/?** displays help for the page heap verification options in GFlags.

Comments

Typing **gflags** without parameters opens the **Global Flags** dialog box.

Typing **gflags /p** without additional parameters displays a list of programs that have page heap verification enabled.

To clear all flags, set *Flag* to **-FFFFFFFF**. (Setting *Flag* to 0 adds zero to the current flag value. It does not clear all flags.)

When you set *Flag* for an image file to **FFFFFFFF**, Windows clears all flags and deletes the GlobalFlag entry in the registry subkey for the image file. The subkey remains.

The **/full**, **/random**, **/size**, **/address**, and **/dlls** parameters for the page heap **/enable** operation determine which allocations are subject to page heap verification and the verification method used. You can use only one of these parameters in each command. The default is standard page heap verification of all allocations of the process. The remaining parameters set options for page heap verification.

The page heap features in GFlags only monitor heap memory allocations that use the standard Windows heap manager functions (**HeapAlloc**, **GlobalAlloc**, **LocalAlloc**, **malloc**, **new**, **new[]**, or their corresponding deallocation functions), or those that use custom operations that call the standard heap manager functions.

To determine whether full or standard page heap verification is enabled for a program, at the command line, type **gflags /p**. In the resulting display, **traces** indicates that standard page heap verification is enabled for the program and **full traces** indicates that full page heap verification is enabled for the program.

The **/enable** parameter sets the Enable page heap (hpa) flag for the image file in the registry. However, the **/enable** parameter turns on *standard* heap verification for the image file by default, unlike the **/i** parameter with the **+hpa** flag, which turns on *full* heap verification for an image file.

Standard page heap verification places random patterns at the end of an allocation and examines the patterns when a heap block is freed. *Full* page heap verification places a zone of reserved virtual memory at the end of each allocation.

Full page heap verification can consume system memory quickly. To enable full page heap verification for memory-intensive processes, use the **/size** or **/dlls** parameter.

After using global flags for debugging, submit a **gflags /p /disable** command to turn off the page heap verification and delete associated registry entries. Otherwise, entries that the debugger reads remain in the

registry. You cannot use the **gflags /i hpa** command for this task, because it turns off page heap verification, but does not delete the registry entries.

By default, on Windows Vista and later versions of Windows, program-specific settings (image file flags and page heap verification settings) are stored in the current user account.

This version of GFlags includes the **-v** options, which enable features being developed for GFlags. However, these features are not yet complete and, therefore, are not documented.

GFlags Flag Table

12/5/2018 • 2 minutes to read • Edit Online

The following table lists the flags that GFlags changes, the hexadecimal value and abbreviation for each flag, and the destination (R for registry, K for kernel, I for image file) in which the flag is valid.

For a detailed description of each flag, see the Global Flag Reference.

For information about using GFlags, see GFlags Overview and GFlags Details.

Important Pool tagging is permanently enabled in Windows Server 2003 and later versions of Windows. On these systems, the **Enable pool tagging** check box on the **Global Flags** dialog box is dimmed, and commands to enable or disable pool tagging fail.

Note The symbolic name of each flag is provided for reference only. Because symbolic names change, they are not a reliable identifier of a global flag.

DESCRIPTION	SYMBOLIC NAME	HEXADECIMAL VALUE	ABBREVIATION	DESTINATION
Buffer DbgPrint Output	FLG_DISABLE_DBGPRINT	0x08000000	ddp	R,K
Create kernel mode stack trace database	FLG_KERNEL_STACK_TRACE_DB	0x2000	kst	R
Create user mode stack trace database	FLG_USER_STACK_TRACE_DB	0x1000	ust	R,K,I
Debug initial command	FLG_DEBUG_INITIAL_COMMAND	0x04	dic	R
Debug WinLogon	FLG_DEBUG_INITIAL_COMMAND_EX	0x04000000	dwl	R
Disable heap coalesce on free	FLG_HEAP_DISABLE_COALESCING	0x00200000	dhc	R,K,I
Disable paging of kernel stacks	FLG_DISABLE_PAGE_KERNEL_STACKS	0x080000	dps	R
Disable protected DLL verification	FLG_DISABLE_PROTDLLS	0x80000000	dpd	R,K,I

DESCRIPTION	SYMBOLIC NAME	HEXADECIMAL VALUE	ABBREVIATION	DESTINATION
Disable stack extension	FLG_DISABLE_STACK_EXTENSION	0x010000	dse	I
Early critical section event creation	FLG_CRITSEC_EVENT_CREATION	0x10000000	cse	R,K,I
Enable application verifier	FLG_APPLICATION_VERIFIER	0x0100	vrf	R,K,I
Enable bad handles detection	FLG_ENABLE_HANDLE_EXCEPTIONS	0x40000000	bhd	R,K
Enable close exception	FLG_ENABLE_CLOSE_EXCEPTIONS	0x400000	ece	R,K
Enable debugging of Win32 subsystem	FLG_ENABLE_CSRDEBUG	0x020000	d32	R
Enable exception logging	FLG_ENABLE_EXCEPTION_LOGGING	0x800000	eel	R,K
Enable heap free checking	FLG_HEAP_ENABLE_FREE_CHECK	0x20	hfc	R,K,I
Enable heap parameter checking	FLG_HEAP_VALIDATE_PARAMETERS	0x40	hpc	R,K,I
Enable heap tagging	FLG_HEAP_ENABLE_TAGGING	0x0800	htg	R,K,I
Enable heap tagging by DLL	FLG_HEAP_ENABLE_TAG_BY_DLL	0x8000	htd	R,K,I
Enable heap tail checking	FLG_HEAP_ENABLE_TAIL_CHECK	0x10	htc	R,K,I
Enable heap validation on call	FLG_HEAP_VALIDATE_ALL	0x80	hvc	R,K,I

DESCRIPTION	SYMBOLIC NAME	HEXADECIMAL VALUE	ABBREVIATION	DESTINATION
Enable loading of kernel debugger symbols	FLG_ENABLE_KDEBUG_SYMBOL_LOAD	0x040000	ksl	R,K
Enable object handle type tagging	FLG_ENABLE_HANDLE_TYPE_TAGGING	0x01000000	eot	R,K
Enable page heap	FLG_HEAP_PAGE_ALLOCS	0x02000000	hpa	R,K,I
Enable pool tagging (Windows 2000 and Windows XP only)	FLG_POOL_ENABLE_TAGGING	0x0400	ptg	R
Enable system critical breaks	FLG_ENABLE_SYSTEM_CRIT_BREAKS	0x100000	scb	R, K, I
Load image using large pages if possible			lpg	I
Maintain a list of objects for each type	FLG_MAINTAIN_OBJECT_TYPELIST	0x4000	otl	R
Enable silent process exit monitoring	FLG_MONITOR_SILENT_PROCESS_EXIT	0x200		R
Object Reference Tracing (Windows Vista and later)				R, K
Show loader snaps	FLG_SHOW_LDR_SNAPS	0x02	sls	R,K,I
Special Pool			spp	R R,K (Windows Vista and later)
Stop on exception	FLG_STOP_ON_EXCEPTION	0x01	soe	R,K,I

DESCRIPTION	SYMBOLIC NAME	HEXADECIMAL VALUE	ABBREVIATION	DESTINATION
Stop on hung GUI	FLG_STOP_ON_HUNG_GUI	0x08	shg	K
Stop on unhandled user-mode exception	FLG_STOP_ON_UNHANDLED_EXCEPTION	0x20000000	sue	R,K,I

GFlags and PageHeap

12/5/2018 • 2 minutes to read • Edit Online

This version of GFlags includes the functionality of PageHeap (pageheap.exe), a tool that enables heap allocation monitoring in Windows. PageHeap enables Windows features that reserve memory at the boundary of each allocation to detect attempts to access memory beyond the allocation.

The page heap options in GFlags let you select *standard heap verification*, which writes fill patterns at the end of each heap allocation and examines the patterns when the allocations are freed, or *full-page heap verification*, which places an inaccessible page at the end of each allocation so that the program stops immediately if it accesses memory beyond the allocation. Because full heap verification uses a full page of memory for each allocation, its widespread use can cause system memory shortages.

- To enable standard page heap verification for all processes, use **gflags /r +hpa** or **gflags /k +hpa**.
- To enable standard page heap verification for one process, use **gflags /p /enable** *ImageFileName*.
- To enable full page heap verification for one process, use **gflags /i** *ImageFileName* **+hpa** or **gflags /p /enable** *ImageFileName* **/full**.

All page heap settings, except for **/k**, are stored in the registry and remain effective until you change them.

Use care in interpreting the **Enable page heap** check box for an image file in the GFlags dialog box. It indicates that page heap verification is enabled for an image file, but it does not indicate whether it is full or standard page heap verification. If the check results from selecting the check box, then full page heap verification is enabled for the image file. However, if the check results from use of the command-line interface, then the check can represent the enabling of either full or standard page heap verification for the image file.

To determine whether full or standard page heap verification is enabled for a program, at the command line, type **gflags /p**. In the resulting display, **traces** indicates that standard page heap verification is enabled for the program and **full traces** indicates that full page heap verification is enabled for the program.

Monitoring Silent Process Exit

12/5/2018 • 5 minutes to read • Edit Online

Beginning with Windows 7, you can use the **Silent Process Exit** tab in GFlags to enter the name of a process that you want to monitor for silent exit.

In the context of this monitoring feature, we use the term *silent exit* to mean that the monitored process terminates in one of the following ways.

Self termination
The monitored process terminates itself by calling **ExitProcess**.

Cross-process termination
A second process terminates the monitored process by calling **TerminateProcess**.

The monitoring feature does not detect normal process termination that happens when the last thread of the process exits. The monitoring feature does not detect process termination that is initiated by kernel-mode code.

To register a process for silent exit monitoring, open the **Silent Process Exit** tab in GFlags. Enter the process name as the **Image** and press the **Tab** key. Check the **Enable Silent Process Exit Monitoring** box, and click **Apply**. This sets the FLG_MONITOR_SILENT_PROCESS_EXIT flag in the following registry entry.

HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options*ProcessName*\GlobalFlag

For more information about this flag, see Enable silent process exit monitoring.

For more information about using the **Silent Process Exit** tab in GFlags, see Configuring Silent Process Exit Monitoring.

In the **Silent Process Exit** tab of GFlags, you can configure the actions that will take place when a monitored process exits silently. You can configure notification, event logging, and creation of dump files. You can specify a process that will be launched when silent exit is detected, and you can specify a list of modules that the monitor will ignore. Several of these settings are available both globally and for individual applications. Global settings apply to all processes that you register for silent exit monitoring. Application settings apply to an individual process and override global settings.

Global settings are stored in the registry under the following key.

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\SilentProcessExit

Application settings are stored in the registry under the following key.

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\SilentProcessExit*ProcessName*

Reporting Mode

The **Reporting Mode** setting is available as an application setting, but not as a global setting. You can use the following check boxes to set the reporting mode.

Launch monitor process Enable dump collection Enable notification The **ReportingMode** registry entry is a bitwise OR of the following flags.

FLAG	VALUE	MEANING
LAUNCH_MONITORPROCESS	0x1	When silent exit is detected, the monitor process (specified in the **Monitor Process** box) is launched.
LOCAL_DUMP	0x2	When silent exit is detected, a dump file is created for the monitored process. In the case of cross-process termination, a dump file is also created for the process that caused the termination.
NOTIFICATION	0x4	When silent exit is detected, a pop-up notification is displayed.

Ignore Self Exits

The **Ignore Self Exits** setting is available as an application setting, but not as a global setting. You can use the **Ignore Self Exits** check box to specify whether self exits are ignored.

The **IgnoreSelfExits** registry entry has one of the following values.

VALUE	MEANING
0x0	Detect and respond to both self termination and cross-process termination.
0x1	Ignore self termination. Detect and respond to cross-process termination.

Monitor Process

You can specify a monitor process by entering a process name, along with command line parameters, in the **Monitor Process** text box. You can use the following variables in your command line.

VARAIBLE	MEANING
%e	ID of the exiting process. This is the monitored process that exited silently.
%i	ID of the initiating process. In the case of self termination, this is the same as the exiting process. In the case of cross-process termination, this is the ID of the process that caused the termination.
%t	ID of the initiating thread. This is the thread that caused the termination.
%c	The status code passed to **ExitThread** or **TerminateThread** .

For example, the following value for **Monitor Process** specifies that on silent exit, WinDbg is launched and attached to the exiting process.

windbg -p %e

The **Monitor Process** command line is stored in the **MonitorProcess** registry entry.

Dump Folder Location

You can use the **Dump folder location** text box to specify a location for the dump files that are written when a silent exit is detected.

The string that you enter for **Dump folder location** is stored in the **LocalDumpFolder** registry entry.

If you do not specify a dump folder location, dump files are written to the default location, which is %TEMP%\Silent Process Exit.

Dump Folder Size

You can use the **Dump folder size** text box to specify the maximum number of dump files that can be written to the dump folder. Enter this value as a decimal integer.

The value that you enter for **Dump folder size** is stored in the **MaximumNumberOfDumpFiles** registry entry.

By default, there is no limit to the number of dump files that can be written.

Dump Type

You can use the **Dump Type** drop-down list to specify the type of dump file (Micro, Mini, Heap, or Custom) that is written when a silent exit is detected.

The dump type is stored in the **DumpType** registry entry, which is a bitwise OR of the members of the **MINIDUMP_TYPE** enumeration. This enumeration is defined in dbghelp.h, which is included in the Debugging Tools for Windows package.

For example, suppose you chose a dump type of **Micro**, and you see that the **DumpType** registry entry has a value of 0x88. The value 0x88 is a bitwise OR of the following two **MINIDUMP_TYPE** enumeration values.

MiniDumpFilterModulePaths	0x00000080
MiniDumpFilterMemory	0x00000008

If you choose a dump type of **Custom**, enter your own bitwise OR of **MINIDUMP_TYPE** enumeration values in the **Custom Dump Type** box. Enter this value as a decimal integer.

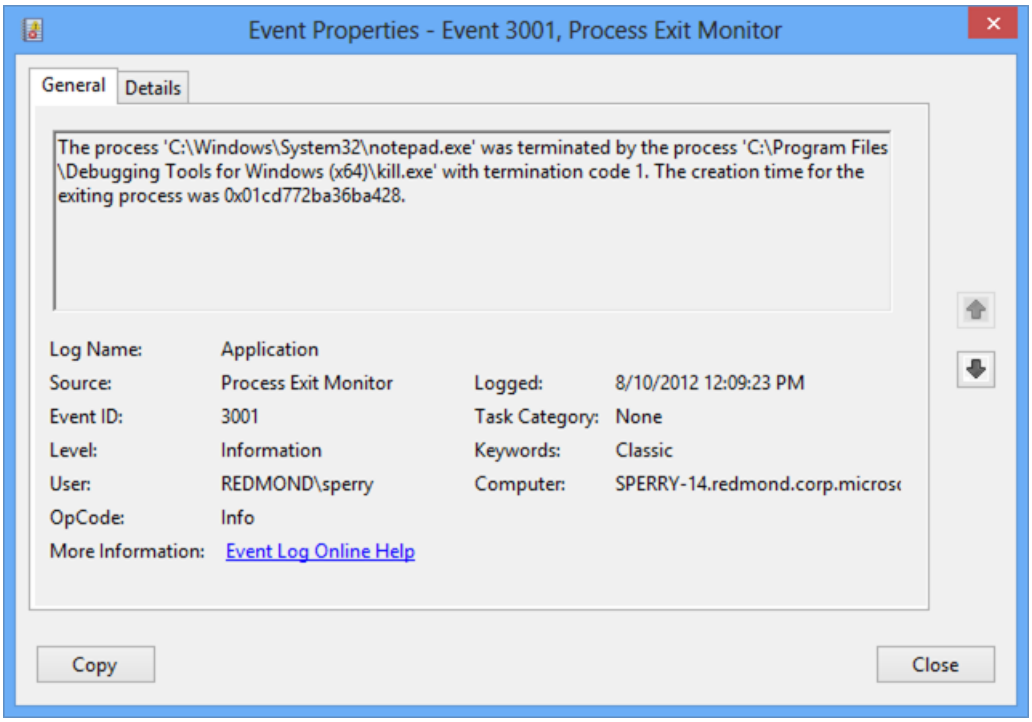
Module Ignore List

You can use the **Module Ignore List** box to specify a list of modules that will be ignored when a silent exit is detected. If the monitored process is terminated by one of the modules in this list, the silent exit is ignored.

The list of modules that you enter in the **Module Ignore List** box is stored in the **ModuleIgnoreList** registry entry.

Reading Process Exit Reports in Event Viewer

When a monitored process exits silently, the monitor creates an entry in Event Viewer. To open Event Viewer, enter the command **eventvwr.msc**. Navigate to **Windows Logs > Application**. Look for log entries that have a **Source** of Process Exit Monitor.


Event Properties - Event 3001, Process Exit Monitor
General
Details
The process 'C:\Windows\System32\notepad.exe' was terminated by the process 'C:\Program Files \Debugging Tools for Windows (x64)\kill.exe' with termination code 1. The creation time for the exiting process was 0x01cd772ba36ba428.
Log Name: Application
Source: Process Exit Monitor
Logged: 8/10/2012 12:09:23 PM
Event ID: 3001
Task Category: None
Level: Information
Keywords: Classic
User: REDMOND\sperry
Computer: SPERRY-14.redmond.corp.micros
OpCode: Info
More Information: Event Log Online Help
Copy
Close

Global Flags Dialog Box

12/5/2018 • 2 minutes to read • Edit Online

You can use the **Global Flags** dialog box to set and clear global flags from a user interface that lists all flags by name. There is no need to look up flag abbreviations or hexadecimal values.

Also, the dialog box provides access to the following features that are not available from the command line:

- **Debugger** – Specifies that a particular program always runs in a debugger.
- **Launch** – Runs a program with the specified debugging settings.
- **Kernel Special Pool Tag** – Configures the Special Pool feature.

This topic includes instructions for the following procedures:

Opening the Dialog Box

Setting and Clearing System-wide Flags

Setting and Clearing Kernel Flags

Setting and Clearing Image File Flags

Launching a Program with Flags

Running a Program in a Debugger

Configuring Special Pool

Configuring Object Reference Tracing

Pool tagging is permanently enabled on Windows Server 2003 and later versions of Windows. On these systems, the **Enable pool tagging** check box on the **Global Flags** dialog box is dimmed, and commands to enable or disable pool tagging fail.

Use care in interpreting the **Enable page heap** check box for an image file in the **Global Flags** dialog box. This check box indicates that page heap verification is enabled for an image file, but it does not indicate whether it is full or standard page heap verification. If the check results from selecting the check box, then full page heap verification is enabled for the image file. However, if the check results from use of the command-line interface, then the check can represent the enabling of either full or standard page heap verification for the image file.

To determine whether a full or standard page heap verification is enabled for a program, at the command line, type **gflags /p**. In the resulting display, **traces** indicates that standard page heap verification is enabled for the program and **full traces** indicates that full page heap verification is enabled for the program.

Opening the Dialog Box

12/5/2018 • 2 minutes to read • Edit Online

To open the Global Flags dialog box

- Double-click the **gflags.exe** icon or, at the command line, or in the **Run** dialog box, type **gflags** (without parameters).
- ○ OR
- Click **Start**, point to **All Programs**, point to **Debugging Tools for Windows,** and then click **Global Flags**.

  On Windows Vista, click **Start**, click **All Programs**, click **Debugging Tools for Windows**, right-click **Global Flags** and then click **Run as administrator**. If you omit this step, Windows displays the **System Registry Gflags Error: 5**. The Gflags dialog box opens, but Gflags commands fail.

Setting and Clearing System-wide Flags

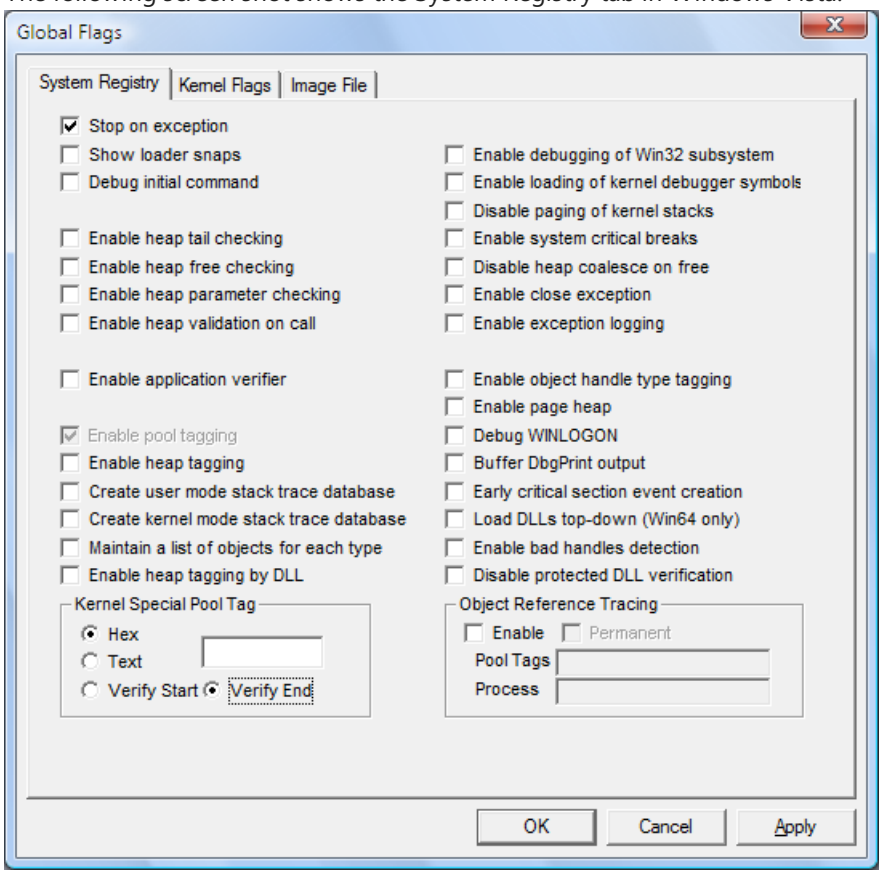
12/5/2018 • 2 minutes to read • Edit Online

System-wide registry settings affect all processes running on Windows. They are saved in the registry and remain effective until you change them.

To set and clear system-wide registry flags

1. Click the **System Registry** tab.

   The following screen shot shows the System Registry tab in Windows Vista.



2. Set or clear a flag by selecting or clearing the check box associated with the flag.
3. When you have selected or cleared all of the flags that you want, click **Apply**.
4. Restart Windows to make the changes effective.

Setting and Clearing Kernel Flags

12/5/2018 • 2 minutes to read • Edit Online

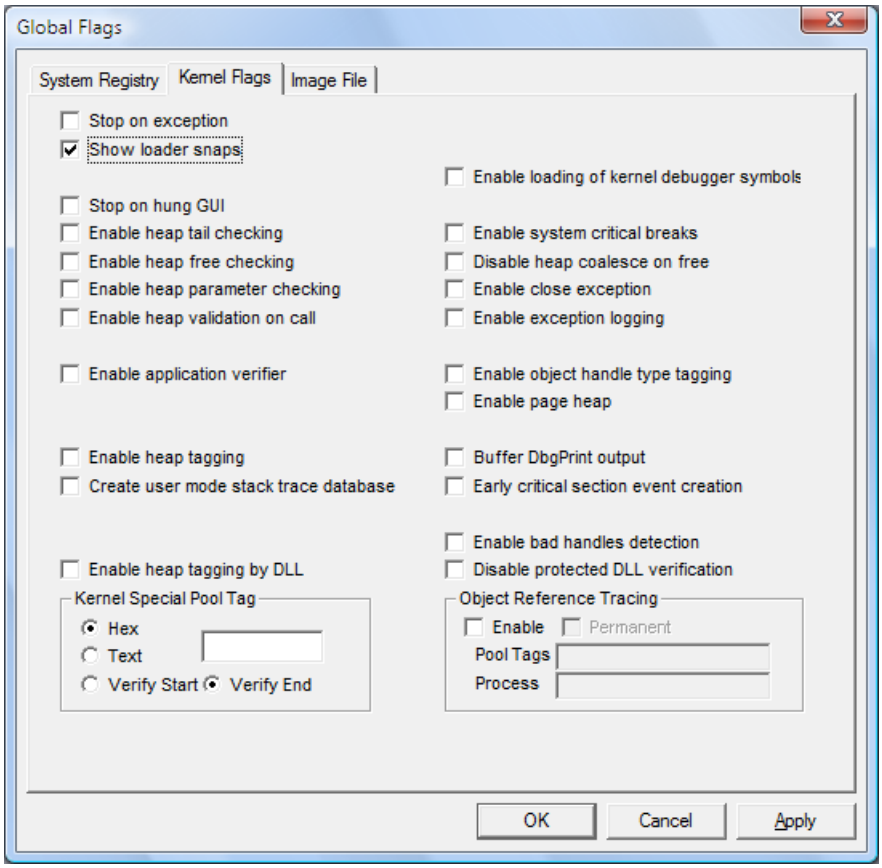
Kernel flag settings, also known as "run-time settings," affect the entire system. They take effect immediately without rebooting, but they are lost if you shut down or restart the system.

Kernel settings take precedence over registry settings during run time, but when you shut down or restart the system, the kernel flag settings are lost and the registry settings are effective again.

To set and clear kernel flags

1. Click the **Kernel Flags** tab.

   The following screen shot shows the **Kernel Flags** tab in Windows Vista.



2. Set or clear a flag by selecting or clearing the check box associated with the flag.

3. When you have selected or cleared all of the flags that you want, click **Apply**.

Setting and Clearing Image File Flags

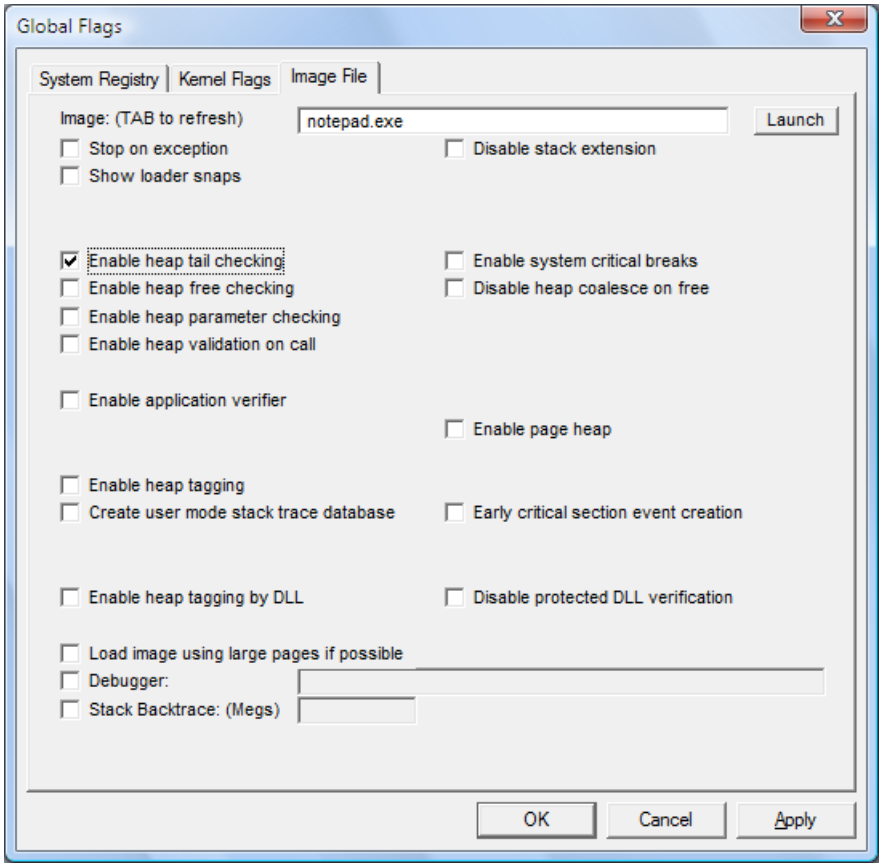
12/5/2018 • 2 minutes to read • Edit Online

Image file settings affect instances of the specified program that start after the command completes. They are saved in the registry and remain effective until you change them.

To set and clear the image file registry flags

1. Click the **Image File** tab.

   The following screen shot shows the **Image File** tab in Windows Vista.



2. In the **Image** box, type the name of an executable file or DLL, including the file name extension,and then press the TAB key.

   This activates the check boxes on the **Image File** tab.

3. Set or clear a flag by selecting or clearing the check box associated with the flag.

4. When you have selected or cleared all of the flags you want, click **Apply**.

Configuring Silent Process Exit Monitoring

12/5/2018 • 2 minutes to read • Edit Online

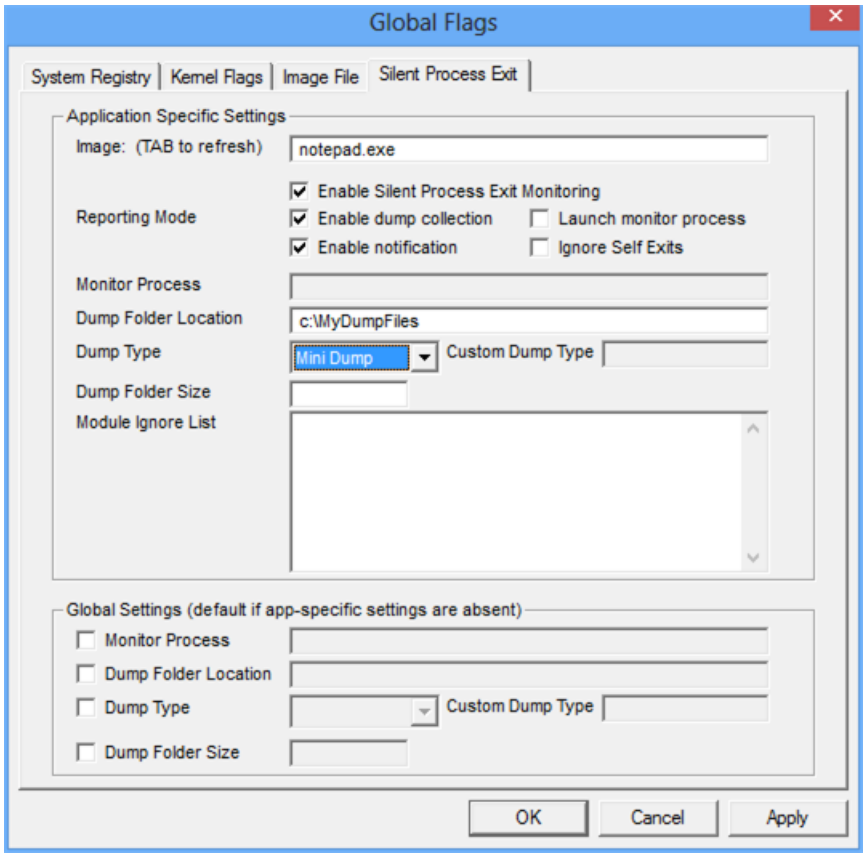
Beginning in Windows 7, you can use the **Silent Process Exit** tab to enable and configure monitoring of silent exit for a process.

Settings that you specify in the **Silent Process Exit** tab are saved in the registry and remain effective until you change them.

To enable and configure silent process exit monitoring

1. Click the **Silent Process Exit** tab.

   The following screen shot shows the **Silent Process Exit** tab in Windows 8.



2. In the **Image** box, type the name of an executable file, including the file name extension, and then press the TAB key.

   This activates the check boxes on the **Silent Process Exit** tab.

3. Specify your preferences by selecting or clearing check boxes and by entering values in text boxes.

4. When you specified all of your preferences, click **Apply**.

Related topics

Monitoring Silent Process Exit

Enable silent process exit monitoring

Launching a Program with Flags

12/5/2018 • 2 minutes to read • Edit Online

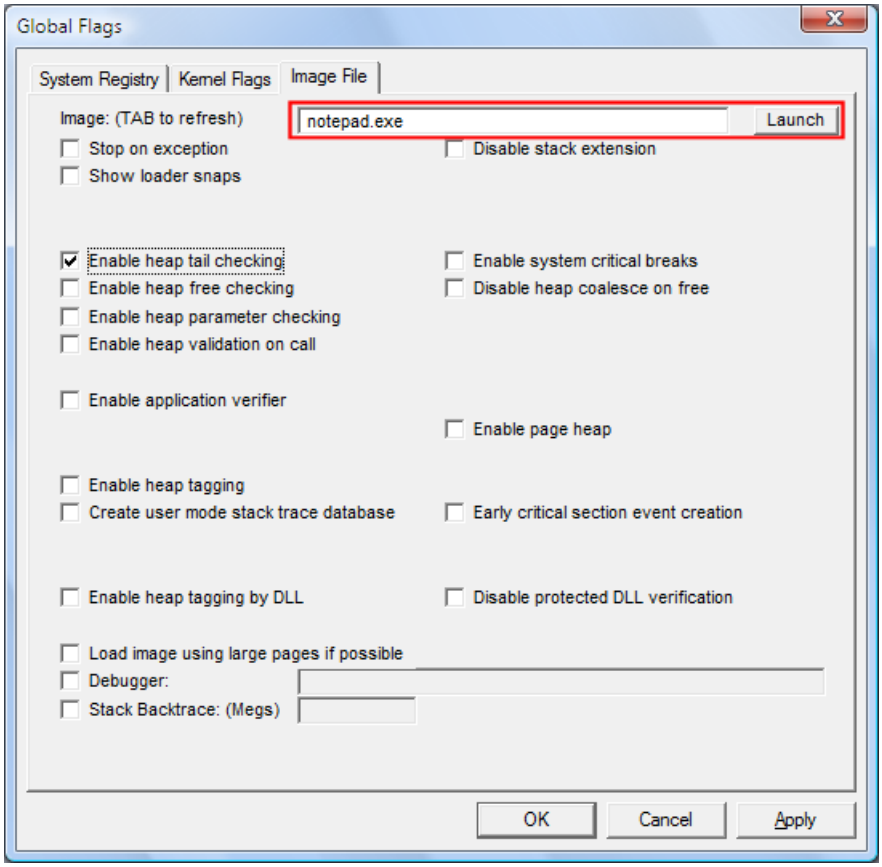
This feature runs a program once with the specified flags. These settings affect only the instance of the program launched. They are not saved in the registry.

To launch a program with flags

1. Click the **Image File** tab.
2. In the **Image** box, type the name of an executable file or DLL, including the file name extension, and any commands for the program, and then press the TAB key.

   This activates the **Launch** button and the check boxes on the **Image File** tab.
3. Set or clear a flag by selecting or clearing the check box associated with the flag.
4. Click the **Launch** button.

   The following screen shot shows the **Launch** button on the **Image File** tab in Windows Vista.



Note Flags set in the registry do not affect the instance of the program that is launched. Flags set in the dialog box are used for the launched instance even when they are not image file flags.

Running a Program in a Debugger

12/5/2018 • 2 minutes to read • Edit Online

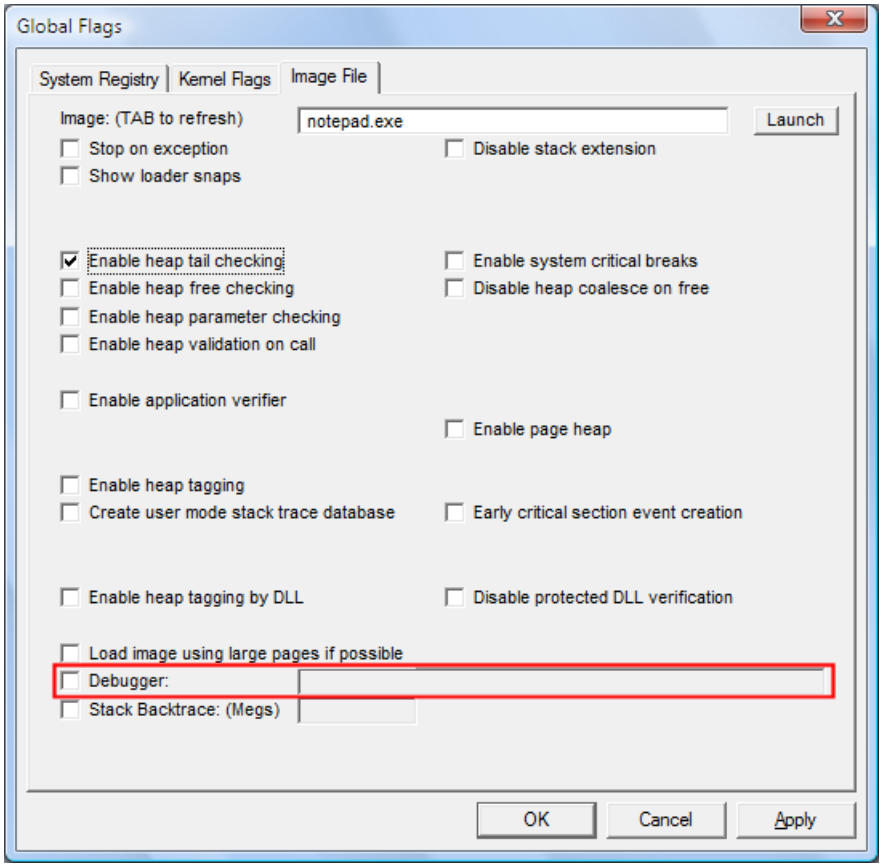
This feature configures the program so that it always runs in a debugger with the specified options. This setting is saved in the registry. It affects all new instances of the program and remains effective until you change it.

To run a program in a debugger

1. Click the **Image File** tab.
2. In the **Image** box, type the name of an executable file or DLL, including the file name extension,and then press the TAB key.

   This activates the check boxes on the **Image File** tab.
3. Click the **Debugger** check box to select it.

   The following screen shot shows the **Debugger** check box on the **Image File** tab in Windows Vista.



4. In the **Debugger** box, type the command to run the debugger, including the path (optional) and name of the debugger and parameters. For example, **ntsd -d -g -G -x** or **c:\debuggers\cdb.exe -g -G**.
5. Click **Apply**.

Configuring Special Pool

12/5/2018 • 2 minutes to read • Edit Online

The Gflags *Special Pool* feature directs Windows to request memory allocations from a reserved memory pool when the memory is allocated with a specified pool tag or is within a specified size range.

For detailed information about this feature, see Special Pool.

In Windows Vista and later versions of Windows, you can configure the Special Pool feature as a system-wide registry setting or as a kernel flags setting that does not require a reboot. In earlier versions of Windows, Special Pool is available only as a registry setting.

In Windows Vista and later versions of Windows, you can also use the command line to request special pool by pool tag. For information, see **GFlags Commands**.

This section includes the following topics.

Requesting Special Pool by Pool Tag

Requesting Special Pool by Allocation Size

Canceling Requests for Special Pool

Detecting Overruns and Underruns

Note Use *Driver Verifier* to request special pool for allocations by a particular driver. For more information, see the "Special Pool" topic in the "Driver Verifier" section of the Windows Driver Kit (WDK).

Requesting Special Pool by Pool Tag

10/8/2019 • 2 minutes to read • Edit Online

You can request special pool for all allocations that use a specified pool tag. Only one pool tag on the system can be associated with kernel special pool requests at one time.

In Windows Vista and later versions of Windows, you can also use the command line to request special pool by pool tag. For information, see **GFlags Commands**.

To request special pool by pool tag

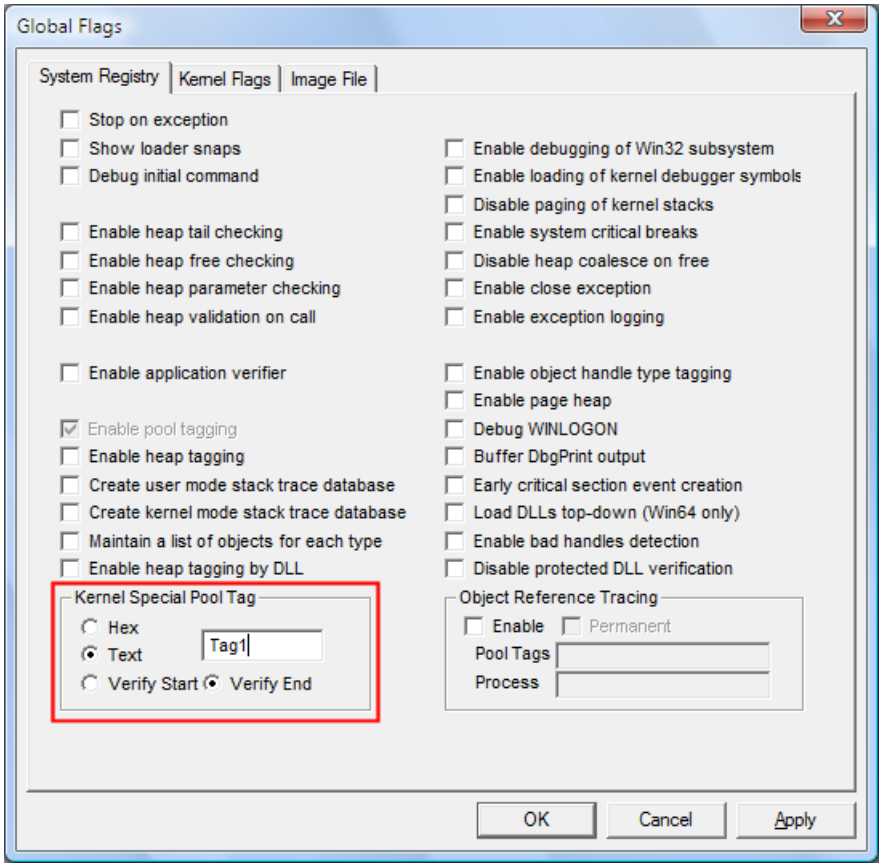
1. Select the **System Registry** tab or the **Kernel Flags** tab.

   On Windows Vista and later versions of Windows, this option is available on both tabs. On earlier versions of Windows, it is available only on the **System Registry** tab.

2. In the **Kernel Special Pool Tag** section, click **Text**, and then type a four-character pattern for the tag.

   The tag can include the **?** (single character) and ***** (multiple characters) wildcard characters. For example, Fat* or Av?4.

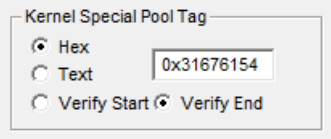
3. The following screen shot shows a tag entered as text on the System Registry tab.



4. Click **Apply**.

   When you click **Apply**, GFlags changes the selection from **Text** to **Hex** and displays the ASCII characters as hexadecimal values in reverse (lower endian) order. For example, if you type **Tag1**, GFlags displays the tag as **0x31676154** (1gaT). This is the way that it is stored in the registry and displayed by the debugger and other tools.

   The following illustration shows the effect of clicking **Apply**.



Remarks

To use this feature effectively, make sure that your driver or other kernel-mode program uses a unique pool tag. If you suspect that your driver is consuming all of the special pool, consider using multiple pool tags in your code. You can then test your driver several times, assigning special pool to one pool tag in each test.

Also, select a pool tag with a hexadecimal value that is greater than the page size of the system. For kernel mode code, if you enter a pool tag that has a value less than PAGE_SIZE, Gflags requests special pool for all allocations whose size is within the corresponding range and requests special pool for allocations with an equivalent pool tag. For example, if you select a size of **30**, special pool will be used for all allocations between 17 and 32 bytes in size, and for allocations with the pool tag 0x0030.

Requesting Special Pool by Allocation Size

12/5/2018 • 2 minutes to read • Edit Online

You can request special pool for allocations within a specified size range.

In Windows Vista and later versions of Windows, you can also use the command line to request special pool by pool tag. For information, see **GFlags Commands**.

Note This method is rarely useful for diagnosing driver errors, because it affects all kernel pool requests of the specified size, regardless of which driver or kernel module requested the allocation.

To request special pool by allocation size

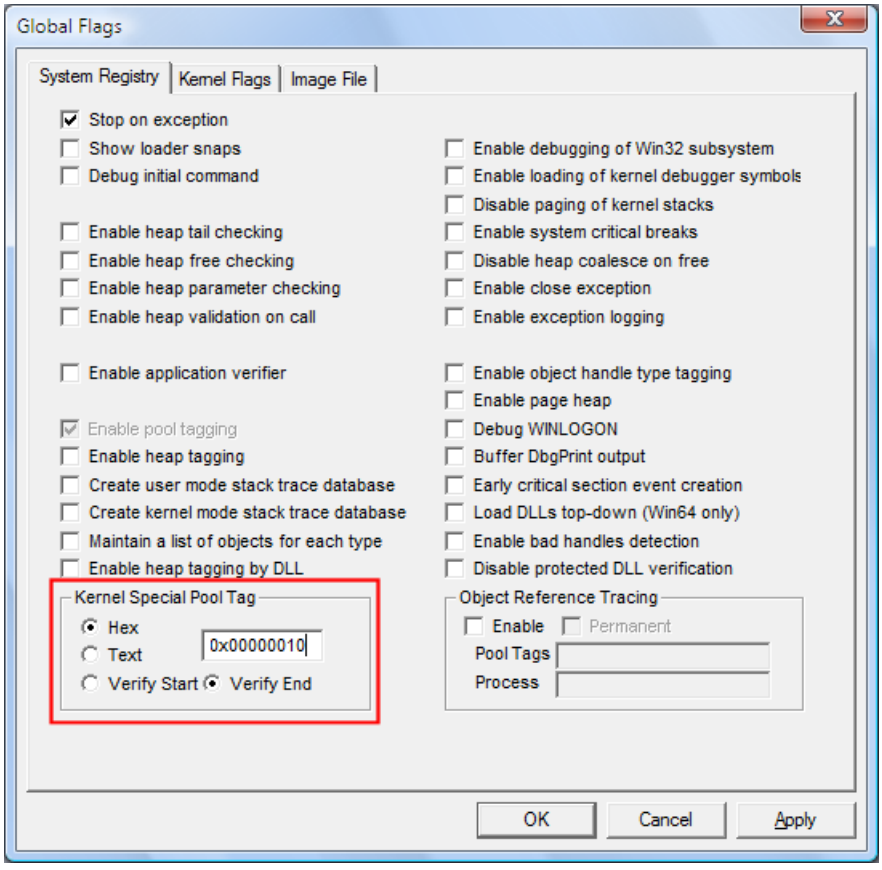
1. Select the **System Registry** tab or the **Kernel Flags** tab.

   On Windows Vista and later versions of Windows, this option is available on both tabs. On earlier versions of Windows, it is available only on the **System Registry** tab.

2. In the **Kernel Special Pool Tag** section, click **Hex**, and then type a number in hexadecimal format that represents a range of sizes. All allocations within this size range will be allocated from special pool. This number must be less than PAGE_SIZE.

3. Click **Apply**.

   The following screen shot shows an allocation size entered as a hexadecimal value.



Canceling Requests for Special Pool

12/5/2018 • 2 minutes to read • Edit Online

You can use GFlags to cancel a request for allocation from the special pool if the request was made by using GFlags. You cannot use GFlags to cancel a request for special pool that was made by using Driver Verifier.

In Windows Vista and later versions of Windows, you can also use the command line to cancel special pool requests. For information, see **GFlags Commands**.

To cancel requests for special pool

1. Select the System Registry tab or the Kernel Flags tab.

   On Windows Vista and later versions of Windows, this option is available on both tabs. On earlier versions of Windows, it is available only on the **System Registry** tab.

2. Delete the text or hexadecimal value from the **Kernel Special Pool Tag** box.

3. Click **Apply**.

Detecting Overruns and Underruns

12/5/2018 • 2 minutes to read • Edit Online

You can use the **Verify Start** or **Verify End** option in GFlags to align allocations from the special pool so that they are best suited to detect overruns (accessing memory past the end of the allocation) or underruns (accessing memory that precedes the beginning of the allocation).

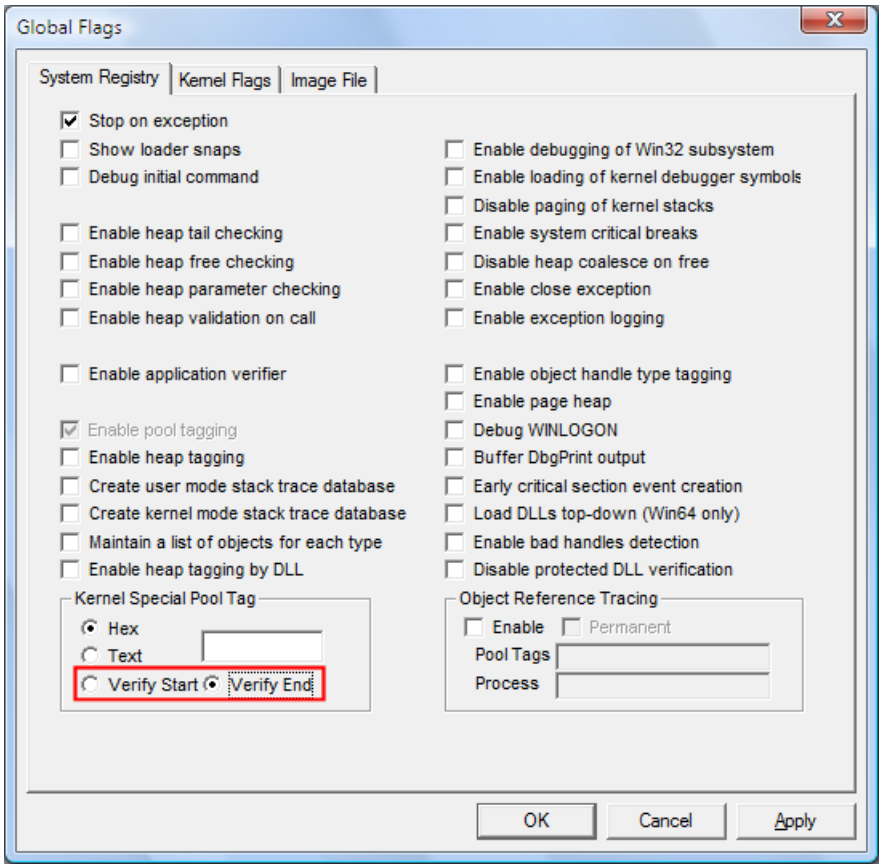
- **Verify Start** enables underrun detection on allocations from the special pool. This causes a bug check when a program tries to access memory preceding its special pool memory allocation.
- **Verify End** enables overrun detection on allocations from the special pool. This causes a bug check when a program tries to access memory beyond its special pool memory allocation. Because overruns are much more common, **Verify End** is the default.

In Windows Vista and later versions of Windows, this option is available on the **System Registry** and **Kernel Flags** tabs. In earlier versions of Windows, it is available only on the **System Registry** tab.

To specify special pool alignment

1. Click the **System Registry** tab.
2. Click **Verify Start** or **Verify End**.
3. Click **Apply**.

   The following screen shot shows the Verify Start and Verify End settings on the System **Registry** tab.



Comments

The **Verify Start** and **Verify End** alignment settings apply to all allocations from the special pool, including special pool allocation requests set in Driver Verifier. If you set the alignment without specifying a pool tag or

allocation size, then the settings apply only to requests set in Driver Verifier.

Configuring Object Reference Tracing

12/5/2018 • 2 minutes to read • Edit Online

You can use Gflags to enable, disable, and configure the Object Reference Tracing feature of Windows. Object Reference Tracing records sequential stack traces whenever an object reference counter is incremented or decremented. The traces can help you to detect object reference errors, including double-dereferencing, failure to reference, and failure to dereference objects. This feature is supported only in Windows Vista and later versions of Windows. For detailed information about this feature, see Object Reference Tracing.

To enable Object Reference Tracing

1. In the Gflags dialog box, select the **System Registry** tab or the **Kernel Flags** tab.
2. In the Object Reference Tracing section, select **Enable**.

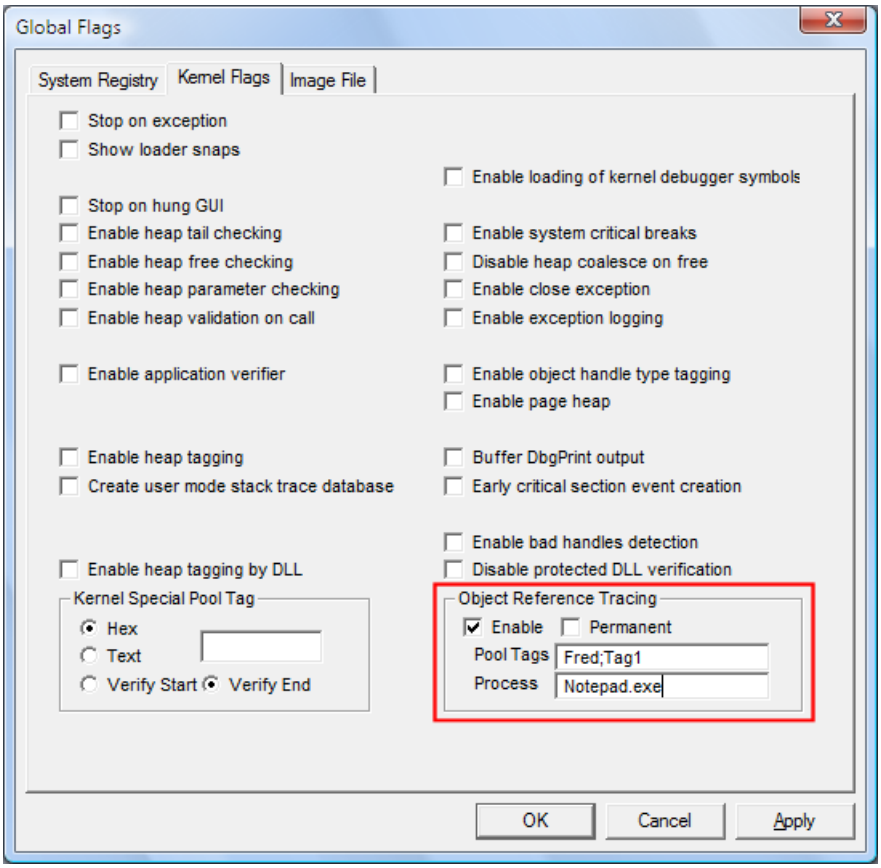
   You must limit the trace to objects with specified pool tags, to objects created by a specified process, or both.
3. To limit the trace to objects with a particular pool tag, type the pool tag name. To list multiple pool tags, use semicolons (;) to separate the pool tags. When you list multiple pool tags, the trace includes objects with any of the specified pool tags. Pool tags are case sensitive.

   For example, Fred;Tag1.
4. To limit the trace to objects that are created by a particular process, type the image name of the process. You can specify only one image file name.

   When you specify both pool tags and a process, the trace includes objects that are created by the process that have any of the specified pool tags.
5. To retain the trace after the trace object is destroyed, select **Permanent**.

   When you select **Permanent**, the trace is retained until you disable object reference tracing, or shut down or restart Windows.
6. Click **Apply** or **OK**.

The following screen shot shows Object Reference Tracing enabled on the **Kernel Flags** tab.



This trace will include only objects that were created by the notepad.exe process that have the pool tag **Fred** or **Tag1**. Because this is a run time (kernel flags) setting, the trace starts immediately. If it were a registry setting, you would have to restart Windows to start the trace.

To disable Object Reference Tracing

1. In the Gflags dialog box, select the **System Registry** tab or the **Kernel Flags** tab. Object Reference Tracing will appear on the latter tab only in Windows Vista and later versions of Windows.
2. In the Object Reference Tracing section, clear the **Enable** check box.

Global Flag Reference

12/5/2018 • 2 minutes to read • Edit Online

This reference describes the flags that GFlags sets.

Note The symbolic name of each flag is provided for reference only. Because symbolic names change, they are not a reliable identifier of a global flag.

The global flags include:

Buffer DbgPrint Output

Create kernel mode stack trace database

Create user mode stack trace database

Debug initial command

Debug WinLogon

Disable heap coalesce on free

Disable paging of kernel stacks

Disable protected DLL verification

Disable stack extension

Early critical section event creation

Enable application verifier

Enable bad handles detection

Enable close exception

Enable debugging of Win32 subsystem

Enable exception logging

Enable heap free checking

Enable heap tagging

Enable heap tagging by DLL

Enable heap tail checking

Enable heap validation on call

Enable loading of kernel debugger symbols

Enable object handle type tagging

Enable page heap

Enable pool tagging

Enable system critical breaks

Load image using large pages if possible

Maintain a list of objects for each type

Object Reference Tracing

Show loader snaps

Special Pool

Stop on exception

Stop on hung GUI

Stop on unhandled user-mode exception

Buffer DbgPrint Output

12/5/2018 • 2 minutes to read • Edit Online

The **Buffer DbgPrint Output** flag suppresses debugger output from **DbgPrint**, **DbgPrintEx**, **KdPrint**, and **KdPrintEx** calls.

Abbreviation	ddp
Hexadecimal value	0x08000000
Symbolic Name	FLG_DISABLE_DBGPRINT
Destination	System-wide registry entry, kernel flag

Comments

When debugger output is suppressed, it does not automatically appear in the kernel debugger. However, the message is always sent to the DbgPrint buffer, where it can be accessed by using the **!dbgprint** debugger extension.

For information about the functions that communicate with the debugger, see Sending Output to the Debugger.

Create kernel mode stack trace database

12/5/2018 • 2 minutes to read • Edit Online

The **Create kernel mode stack trace database** flag creates a run-time stack trace database of kernel operations, such as resource objects and object management operations, and works only when using the checked build of Windows.

Abbreviation	kst
Hexadecimal value	0x2000
Symbolic Name	FLG_KERNEL_STACK_TRACE_DB
Destination	System-wide registry entry

Comments

GFlags displays this flag as a kernel flag setting, but it is not effective at run time, because the kernel is already started.

Create user mode stack trace database

12/5/2018 • 2 minutes to read • Edit Online

The **Create user mode stack trace database** flag creates a run-time stack trace database in the address space of a particular process (image file mode) or all processes (system-wide).

Abbreviation	ust
Hexadecimal value	0x1000
Symbolic Name	FLG_USER_STACK_TRACE_DB
Destination	System-wide registry entry, kernel flag, image file registry entry

Debug initial command

12/5/2018 • 2 minutes to read • Edit Online

The **Debug initial command** flag debugs the Client Server Run-time Subsystem (CSRSS) and the WinLogon process.

Abbreviation	dic
Hexadecimal value	0x4
Symbolic Name	FLG_DEBUG_INITIAL_COMMAND
Destination	System-wide registry entry, kernel flag

Comments

NTSD debugs the processes (using the command **ntsd -d**), but control is redirected to the kernel debugger.

For details on NTSD, see Debugging Using CDB and NTSD.

See Also

Enable debugging of Win32 subsystem

Debug WinLogon

12/5/2018 • 2 minutes to read • Edit Online

The **Debug WinLogon** flag debugs the WinLogon service.

Abbreviation	dwl
Hexadecimal value	0x04000000
Symbolic Name	FLG_DEBUG_INITIAL_COMMAND_EX
Destination	System-wide registry entry

Comments

NTSD debugs Winlogon (by using the command **ntsd -d -g -x**), but control is redirected to the kernel debugger.

For details on NTSD, see Debugging Using CDB and NTSD.

See Also

Debug initial command, Enable debugging of Win32 subsystem

Disable heap coalesce on free

12/5/2018 • 2 minutes to read • Edit Online

The **Disable heap coalesce on free** flag leaves adjacent blocks of heap memory separate when they are freed.

Abbreviation	dhc
Hexadecimal value	0x00200000
Symbolic Name	FLG_HEAP_DISABLE_COALESCING
Destination	System-wide registry entry, kernel flag, image file registry entry

Comments

By default, Windows combines ("coalesces") newly-freed adjacent blocks into a single block. Combining the blocks takes time, but reduces fragmentation that might force the heap to allocate additional memory when it cannot find contiguous memory.

This flag is used for maintaining compatibility with old applications.

Disable paging of kernel stacks

12/5/2018 • 2 minutes to read • Edit Online

The **Disable paging of kernel stacks** flag prevents paging of the kernel-mode stacks of inactive threads.

Abbreviation	dps
Hexadecimal value	0x80000
Symbolic Name	FLG_DISABLE_PAGE_KERNEL_STACKS
Destination	System-wide registry entry

Comments

Generally, the kernel-mode stack cannot be paged; it is guaranteed to be resident in memory. However, Windows occasionally pages the kernel stacks of inactive threads. This flag prevents these occurrences.

The kernel debugger can provide information about a thread only when its stack is in physical memory. This flag is particularly important when debugging deadlocks and in other cases when every thread must be tracked.

Disable protected DLL verification

12/5/2018 • 2 minutes to read • Edit Online

The **Disable protected DLL verification** flag appears in GFlags, but it has no effect on Windows.

Abbreviation	dpd
Hexadecimal value	0x80000000
Symbolic Name	FLG_DISABLE_PROTDLLS
Destination	System-wide registry entry, kernel flag, image file registry entry

Disable stack extension

12/5/2018 • 2 minutes to read • Edit Online

The **Disable stack extension** flag prevents the kernel from extending the stacks of the threads in the process beyond the initial committed memory.

Abbreviation	dse
Hexadecimal value	0x10000
Symbolic Name	FLG_DISABLE_STACK_EXTENSION
Destination	Image file registry entry

Comments

This feature is used to simulate low memory conditions (where stack extensions fail) and to test the strategic system processes that are expected to run well even with low memory.

Early critical section event creation

12/5/2018 • 2 minutes to read • Edit Online

The **Early critical section event creation** flag creates event handles when a critical section is initialized, rather than waiting until the event is needed.

Abbreviation	cse
Hexadecimal value	0x10000000
Symbolic Name	FLG_CRITSEC_EVENT_CREATION
Destination	System-wide registry entry, kernel flag, image file registry entry

Comments

When Windows cannot create an event, it generates the exception during initialization and the calls to enter and leave the critical section do not fail.

Because this flag uses a significant amount of nonpaged pool memory, use it only on very reliable systems that have sufficient memory.

Enable application verifier

6/7/2019 • 2 minutes to read • Edit Online

The **Enable application verifier** flag enables system features that are used for user-mode application testing, such as page heap verification, lock checks, and handle checks.

Abbreviation	vrf
Hexadecimal value	0x100
Symbolic Name	FLG_APPLICATION_VERIFIER
Destination	System-wide registry entry, kernel flag, image file registry entry

Comments

This flag enables only the most basic detection features. To test user-mode applications reliably, use Application Verifier (appverif.exe). For more information, see Application Verifier.

Enable bad handles detection

12/5/2018 • 2 minutes to read • Edit Online

The **Enable bad handles detection** flag raises a user-mode exception (STATUS_INVALID_HANDLE) whenever a user-mode process passes an invalid handle to the Object Manager.

Abbreviation	bhd
Hexadecimal value	0x40000000
Symbolic Name	FLG_ENABLE_HANDLE_EXCEPTIONS
Destination	System-wide registry entry, kernel flag

Enable close exception

12/5/2018 • 2 minutes to read • Edit Online

The **Enable close exception** flag raises a user-mode exception whenever an invalid handle is passed to the **CloseHandle** interface or related interfaces, such as **SetEvent**, that take handles as arguments.

Abbreviation	ece
Hexadecimal value	0x00400000
Symbolic Name	FLG_ENABLE_CLOSE_EXCEPTIONS
Destination	System-wide registry entry, kernel flag

Note This flag is still supported, but the Enable bad handles detection flag (bhd), which performs a more comprehensive check of handle use, is preferred.

Enable debugging of Win32 subsystem

12/5/2018 • 2 minutes to read • Edit Online

The **Enable debugging of Win32 subsystem** flag debugs the Client Server Run-time Subsystem (csrss.exe) in the NTSD debugger.

Abbreviation	d32
Hexadecimal value	0x20000
Symbolic Name	FLG_ENABLE_CSRDEBUG
Destination	System-wide registry entry

Comments

NTSD debugs the process by using the command **ntsd -d -p -1**.

This flag is effective only when the Debug initial command flag (dic) or the Debug WinLogon flag (dwl) is also set.

For details on NTSD, see Debugging Using CDB and NTSD.

Enable exception logging

12/5/2018 • 2 minutes to read • Edit Online

The **Enable exception logging** flag creates a log of exception records in the kernel run-time library. You can access the log from a kernel debugger.

Abbreviation	eel
Hexadecimal value	0x00800000
Symbolic Name	FLG_ENABLE_EXCEPTION_LOGGING
Destination	System-wide registry entry, kernel flag

Enable heap free checking

12/5/2018 • 2 minutes to read • Edit Online

The **Enable heap free checking** flag validates each heap allocation when it is freed.

Abbreviation	hfc
Hexadecimal value	0x20
Symbolic Name	FLG_HEAP_ENABLE_FREE_CHECK
Destination	System-wide registry entry, kernel flag, image file registry entry

See Also

Enable heap tail checking, Enable heap parameter checking

Enable heap parameter checking

12/5/2018 • 2 minutes to read • Edit Online

The **Enable heap parameter checking** flag verifies selected aspects of the heap whenever a heap function is called.

Abbreviation	hpc
Hexadecimal value	0x40
Symbolic Name	FLG_HEAP_VALIDATE_PARAMETERS
Destination	System-wide registry entry, kernel flag, image file registry entry

See Also

Enable heap validation on call

Enable heap tagging

12/5/2018 • 2 minutes to read • Edit Online

The **Enable heap tagging** flag assigns unique tags to heap allocations.

Abbreviation	htg
Hexadecimal value	0x800
Symbolic Name	FLG_HEAP_ENABLE_TAGGING
Destination	System-wide registry entry, kernel flag, image file registry entry

Comments

You can display the tag by using the **!heap** debugger extension with the -t parameter.

See Also

Enable heap tagging by DLL

Enable heap tagging by DLL

12/5/2018 • 2 minutes to read • Edit Online

The **Enable heap tagging by DLL** flag assigns a unique tag to heap allocations created by the same DLL.

Abbreviation	htd
Hexadecimal value	0x8000
Symbolic Name	FLG_HEAP_ENABLE_TAG_BY_DLL
Destination	System-wide registry entry, kernel flag, image file registry entry

Comments

You can display the tag by using the **!heap** debugger extension with the -t parameter.

See Also

Enable heap tagging

Enable heap tail checking

12/5/2018 • 2 minutes to read • Edit Online

The **Enable heap tail checking** flag checks for buffer overruns when the heap is freed.

Abbreviation	htc
Hexadecimal value	0x10
Symbolic Name	FLG_HEAP_ENABLE_TAIL_CHECK
Destination	System-wide registry entry, kernel flag, image file registry entry

Comments

This flag adds a short pattern to the end of each allocation. The Windows heap manager detects the pattern when the block is freed and, if the block was modified, the heap manager breaks into the debugger.

See Also

Enable heap free checking, Enable heap parameter checking

Enable heap validation on call

12/5/2018 • 2 minutes to read • Edit Online

The **Enable heap validation on call** flag validates the entire heap each time a heap function is called.

Abbreviation	hvc
Hexadecimal value	0x80
Symbolic Name	FLG_HEAP_VALIDATE_ALL
Destination	System-wide registry entry, kernel flag, image file registry entry

Comments

To avoid the high overhead associated with this flag, use the **HeapValidate** function instead of setting this flag, especially at critical junctures, such as when the heap is destroyed. However, this flag is useful for detecting random corruption in a pool.

See Also

Enable heap parameter checking

Enable loading of kernel debugger symbols

12/5/2018 • 2 minutes to read • Edit Online

The **Enable loading of kernel debugger symbols** flag loads kernel symbols into the kernel memory space the next time Windows starts.

Abbreviation	ksl
Hexadecimal value	0x40000
Symbolic Name	FLG_ENABLE_KDEBUG_SYMBOL_LOAD
Destination	System-wide registry entry, kernel flag

Comments

The kernel symbols are used in kernel profiling and by advanced kernel debugging tools.

Enable object handle type tagging

12/5/2018 • 2 minutes to read • Edit Online

The **Enable object handle type tagging** flag appears in GFlags, but it has no effect on Windows.

Abbreviation	eot
Hexadecimal value	0x01000000
Symbolic Name	FLG_ENABLE_HANDLE_TYPE_TAGGING
Destination	System-wide registry entry, kernel flag

Enable page heap

12/5/2018 • 2 minutes to read • Edit Online

The **Enable page heap** flag turns on page heap verification, which monitors dynamic heap memory operations, including allocate and free operations, and causes a debugger break when the verifier detects a heap error.

Abbreviation	hpa
Hexadecimal value	0x02000000
Symbolic Name	FLG_HEAP_PAGE_ALLOCS
Destination	System-wide registry entry, kernel flag, image file registry entry

Comments

This option enables full page heap verification when set for image files and standard page heap verification when set in the system registry or as a kernel flag.

- *Full page heap verification* (for **/i**) places a zone of reserved virtual memory at the end of each allocation.
- *Standard page heap verification* (for **/r** or **/k**) places random patterns at the end of an allocation and examines the patterns when a heap block is freed.

Setting this flag for an image file is the same as typing **gflags /p /enable** *ImageFile* **/full** for the image file at the command line.

Enable pool tagging

12/5/2018 • 2 minutes to read • Edit Online

The **Enable pool tagging** flag collects data and calculates statistics about pool memory allocations sorted by pool tag value.

Abbreviation	ptg
Hexadecimal value	0x400
Symbolic Name	FLG_POOL_ENABLE_TAGGING
Destination	System-wide registry entry

Comments

This flag is permanently set in Windows Server 2003 and later versions of Windows. On these systems, the **Enable pool tagging** check box in the Global Flags dialog box is dimmed and commands to enable or disable pool tagging fail.

Use **ExAllocatePoolWithTag** or **ExAllocatePoolWithQuotaTag** to set the tag value. When no tag value is specified (**ExAllocatePool**, **ExAllocatePoolWithQuota**), Windows creates a tag with the default value of "None." Because data for all allocations with a "None" tag is combined, you cannot distinguish the data for a specific allocation. For information about these routines, see the Windows Driver Kit (WDK).

In Windows XP and earlier systems, this flag also directs Windows to attach a pool tag even when the pool memory is allocated by using **ExAllocatePoolWithQuotaTag**. Otherwise, the tag bytes are used to store the quota values. In Windows Server 2003, tag values and quota values are stored in separate fields that are attached to every pool memory allocation.

Note To display the data that Windows collects about a tagged allocation, use PoolMon, a tool that is included in the Windows Driver Kit. The description of the **Enable Pool Tagging** flag in the Windows XP Support Tools documentation is incomplete. This flag directs Windows to collect and process data by tag value.

Enable silent process exit monitoring

12/5/2018 • 2 minutes to read • Edit Online

The **Enable silent process exit monitoring** flag enables silent exit monitoring for a process.

Hexadecimal value	0x200
Symbolic Name	FLG_MONITOR_SILENT_PROCESS_EXIT
Destination	Image file registry entry

Comments

For more information about monitoring a process for silent exit, see Monitoring Silent Process Exit.

Enable system critical breaks

12/5/2018 • 2 minutes to read • Edit Online

The **Enable system critical breaks** flag forces a system break into the debugger.

Abbreviation	scb
Hexadecimal value	0x100000
Symbolic Name	FLG_ENABLE_SYSTEM_CRIT_BREAKS
Destination	System-wide registry entry, kernel flag, image file registry entry

Comments

When set for a process (image file), this flag forces a system break into the debugger whenever the specified process stops abnormally. This flag is effective only when the process calls the **RtlSetProcessBreakOnExit** and **RtlSetThreadBreakOnExit** interfaces.

When set system-wide (registry or kernel flag), this flag forces a system break into the debugger whenever processes that have called the **RtlSetProcessBreakOnExit** and **RtlSetThreadBreakOnExit** interfaces stop abnormally.

Load image using large pages if possible

12/5/2018 • 2 minutes to read • Edit Online

The **Load image using large pages if possible** setting directs the system to use large pages (4 MB) rather than the standard small pages (4 KB) when mapping binaries into the process address space.

This setting is most helpful for large executable files, because it significantly reduces the number of page table entries in physical memory.

Abbreviation	lpg
Hexadecimal value	(none)
Symbolic Name	
Destination	Image file registry entry

Comments

This setting is not technically a global flag, because its value is stored in a separate registry entry, not as a value of the GlobalFlag registry entry. As a result, you cannot set it by using a hexadecimal value, and when you select this setting for an image file, it appears in the **Other settings** field of the display.

Maintain a list of objects for each type

12/5/2018 • 2 minutes to read • Edit Online

The **Maintain a list of objects for each type** flag collects and maintains a list of active objects by object type, for example, event, mutex, and semaphore.

Abbreviation	otl
Hexadecimal value	0x4000
Symbolic Name	FLG_MAINTAIN_OBJECT_TYPELIST
Destination	System-wide registry entry

Comments

To display the object list, use the sysinternals tool Handle.

Note The linked lists created when you set this flag use eight bytes of overhead for each object. Remember to clear this flag when your analysis is complete.

Object Reference Tracing

12/5/2018 • 2 minutes to read • Edit Online

The **Object Reference Tracing** feature records sequential stack traces each time that an object reference counter is incremented or decremented. The traces can help you to detect object reference errors, including double-dereferencing, failure to reference, and failure to dereference objects. This feature is supported only in Windows Vista and later versions of Windows.

For information about configuring the Object Reference Tracing feature in the **Global Flags** dialog box, see Configuring Object Reference Tracing. For information about configuring the Object Reference Tracing feature at the command prompt, see **GFlags Commands**. For an example, see Example 15: Using Object Reference Tracing.

Object reference traces are most useful when you suspect that a particular object is not being referenced or dereferenced properly, typically because increased pool usage indicates that an object is leaking, or a process or session cannot be ended, even though its handle count is zero. Unlike traces that are recorded in logs for later review, object reference traces are designed to be used in real time, while the process is running and the object is being referenced and dereferenced. You view an object reference trace in the debugger by using the **!obtrace debugger extension**. Because this extension requires a specified object address, you must know in advance which object is the likely source of the error.

The following rules apply to Object Reference Tracing:

- You can run only one object reference trace at a time.
- Because a kernel-wide trace is not practical, you must limit the trace to objects that are created with specified pool tags, or to objects that are created by a specified process (indicated by an image file name), or both.
- You can specify only one image file for each trace. If you specify an image file, the trace is limited to objects that are created by the processes that the image represents. Objects that are referenced by the process, but are created by a different process, are not traced.
- You can specify a maximum of 16 pool tags for each trace. Objects with any of the specified pool tags are traced.
- If you specify both an image file and one or more pool tags, the trace is limited to objects that are created by the process and have any of the specified pool tags.
- Object Reference Tracing cannot trace processes that are already running when a trace is started. The trace includes only the objects of processes that start after the trace begins.
- Objects marked for tracing are traced until the object is destroyed or tracing is disabled. By default, the traces for an object are maintained only until the object is destroyed, but you can specify a "permanent" trace (**/p**) where the trace is retained until tracing is disabled.
- You can store the Object Reference Tracing configuration as a registry setting or a kernel flag (run-time) setting. If you have both registry and kernel flag settings, the run-time settings take precedence, but are lost when you shut down or restart the computer.

Show loader snaps

12/5/2018 • 2 minutes to read • Edit Online

The **Show loader snaps** flag captures detailed information about the loading and unloading of executable images and their supporting library modules and displays the data in the kernel debugger console.

Abbreviation	sls
Hexadecimal value	0x2
Symbolic Name	FLG_SHOW_LDR_SNAPS
Destination	System-wide registry entry, kernel flag, image file registry entry

Comments

For system-wide (registry or kernel flag), this flag displays information about driver loading and unloading operations.

For per-process (image file), this flag displays information about loading and unloading of DLLs.

Special Pool

7/9/2019 • 4 minutes to read • Edit Online

The **Special Pool** feature configures Windows to request memory allocations from a reserved memory pool when the memory is allocated with a specified pool tag or is within a specified size range.

Abbreviation	spp
Hexadecimal value	(None)
Symbolic Name	(None)
Destination	System-wide registry entry (Windows Vista and later) System-wide registry entry, kernel flag

Selecting a Pool Tag

When requesting special pool for a particular pool tag, make sure that your driver or other kernel-mode program uses a unique pool tag.

Also, when creating a pool tag (such as by using **ExAllocatePoolWithTag**), consider entering the tag characters in reverse order. For example, if the tag is **Fred**, consider entering it as **derF** (0x64657246). Pool tags are stored in the registry and displayed in the debugger and other tools in reverse (lower endian) order. If you enter them in reverse order, they are displayed in forward order (0x46726564)

If you suspect that your driver is consuming all of the special pool, consider using multiple pool tags in your code. You can then test your driver several times, assigning special pool to one pool tag in each test.

Also, select a pool tag with a hexadecimal value that is greater than the page size of the system. For kernel mode code, if you enter a pool tag that has a value less than PAGE_SIZE, Gflags requests special pool for all allocations whose size is within the corresponding range and requests special pool for allocations with an equivalent pool tag. For example, if you select a size of **30**, special pool will be used for all allocations between 17 and 32 bytes in size, and for allocations with the pool tag 0x0030.

Selecting an Allocation Size

Use the following guidelines to select an allocation size for the Special Pool feature.

On a computer with an x86 processor, PAGE_SIZE is 0x1000 and the allocation size ranges are 8 bytes in length. To configure the Special Pool feature for all allocations with sizes in this range, enter a number equal to the maximum of this range plus 8. (This number is always a multiple of 8.) The following table illustrates these values:

SIZE RANGE	ENTER THIS NUMBER
1 to 8 bytes	10 (decimal 16)

SIZE RANGE	ENTER THIS NUMBER
9 to 16 bytes	18 (decimal 24)
17 to 24 bytes	20 (decimal 32)
...	...
0xFE9 to 0xFF0 bytes	FF8 (decimal 4088)

On a computer with an AMD x86-64 processor, PAGE_SIZE is 0x1000 and the allocation size ranges are 16 bytes in length. To configure the Special Pool feature for all allocations with sizes in this range, enter a number equal to the maximum of this range plus 16. (This number is always a multiple of 16.) The following table illustrates these values:

SIZE RANGE	ENTER THIS NUMBER
1 to 16 bytes	20 (decimal 32)
17 to 32 bytes	30 (decimal 48)
33 to 48 bytes	40 (decimal 64)
...	...
0xFD1 to 0xFE0 bytes	FF0 (decimal 4080)

On a computer with any processor, you can use an asterisk (*****) or 0x2A (decimal 42) to configure the Special Pool feature for all memory allocations on the system.

Comments

For information about configuring the Special Pool feature in the Global Flags Dialog Box, see Configuring Special Pool. For information about configuring the Special Pool feature at the command line, see **GFlags Commands**. For an example, see Example 14: Configuring Special Pool.

The Special Pool feature of Gflags directs Windows to request memory allocations from a reserved memory pool when the memory is allocated with a specified pool tag or is within a specified size range. To request special pool for all allocations by a particular driver, use Driver Verifier. For more information, see the "Special Pool" topic in the "Driver Verifier" section of the Windows Driver Kit (WDK).

The special pool features of Gflags and Driver Verifier help you to detect and identify the source of errors in kernel pool use, such as writing beyond the allocated memory space, or referring to memory that has already been freed.

Not all special pool requests are fulfilled. Each allocation from the special pool uses one page of nonpageable physical memory and two pages of virtual address space. If the special pool is exhausted, memory is allocated from the standard pool until the special pool becomes available again. When a special pool request is filled from the standard pool, the requesting function returns a success status. It does not return an error, because the

allocation has succeeded, even though it was not filled from special pool.

The size of the special pool increases with the amount of physical memory on the system; ideally this should be at least 1 Gigabyte (GB). On x86 machines, because virtual (in addition to physical) space is consumed, do not use the **/3GB** boot option when using special pool. It is also a good idea to increase the pagefile minimum/maximum quantities by a factor of two or three.

You can also configure the Special Pool feature to align memory allocation to detect references to memory preceding the allocation ("underruns") or references to memory beyond the allocation ("overruns"). This feature is available only in the Global Flags dialog box on all versions of Windows. For details, see Detecting Overruns and Underruns.

On Windows Vista and later versions of Windows, you can configure the Special Pool feature as a registry setting that requires a reboot, but remains effective until you change it, or as a kernel flag setting that does not require a reboot, but is effective only until you reboot or shut down Windows. In earlier versions of Windows, Special Pool is only available as a registry setting.

On Windows Vista and later versions of Windows, you can configure the Special Pool feature either by using the Global Flags dialog box or at the command line. In earlier version of Windows, this feature is available only in the Global Flags dialog box.

Stop on exception

12/5/2018 • 2 minutes to read • Edit Online

The **Stop on exception** flag causes the kernel to break into the kernel debugger whenever a kernel-mode exception occurs.

Abbreviation	soe
Hexadecimal value	0x1
Symbolic Name	FLG_STOP_ON_EXCEPTION
Destination	System-wide registry entry, kernel flag, image file registry entry

Comments

Windows passes all first chance exceptions (except for STATUS_PORT_DISCONNECT) with a severity of Warning or Error to the debugger before passing them to a local exception handler.

Stop on hung GUI

12/5/2018 • 2 minutes to read • Edit Online

The **Stop on hung GUI** flag appears in GFlags, but it has no effect on Windows.

Abbreviation	shg
Hexadecimal value	0x8
Symbolic Name	FLG_STOP_ON_HUNG_GUI
Destination	Kernel flag

Stop on unhandled user-mode exception

12/5/2018 • 2 minutes to read • Edit Online

The **Stop on unhandled user-mode exception** flag causes a break into the kernel debugger whenever an unhandled user-mode exception occurs.

Abbreviation	sue
Hexadecimal value	0x20000000
Symbolic Name	FLG_STOP_ON_UNHANDLED_EXCEPTION
Destination	System-wide registry entry, kernel flag, image file registry entry

GFlags Examples

12/5/2018 • 2 minutes to read • Edit Online

The following examples show how to submit GFlags commands and how to use GFlags features in practical scenarios.

This section includes the following topics.

Example 1: Displaying Global Flags

Example 2: Setting a Flag by Using a Flag Abbreviation

Example 3: Setting a Flag by Using Its Hexadecimal Value

Example 4: Setting Multiple Flags

Example 5: Clearing a Flag

Example 6: Clearing All Flags

Example 7: Clearing All Flags for an Image File

Example 8: Enlarging the User-Mode Stack Trace Database

Example 9: Detecting a Pool Memory Leak

Example 10: Detecting a Heap Memory Leak in a Process

Example 11: Enabling Page Heap Verification

Example 12: Using Page Heap Verification to Find a Bug

Example 13: Listing Image Files with Global Flags

Example 14: Configuring Special Pool

Example 15: Using Object Reference Tracing

The examples in the second section apply to features available only in Windows Vista and later versions of Windows.

Example 1: Displaying Global Flags

12/5/2018 • 2 minutes to read • Edit Online

The commands demonstrated in this example display the system-wide flags set in the registry, the system flags set for the session (kernel mode), and the flags set for an image file.

The following GFlags command displays the current value of the system-wide flags set in the registry. It uses the **/r** parameter to specify the system-wide registry entry.

```
gflags /r
```

In response, GFlags displays a single hexadecimal value representing the sum of all flags set and a list of the flags set.

```
Current Boot Registry Settings are: 40001400
    ptg - Enable pool tagging
    ust - Create user mode stack trace database
    bhd - Enable bad handles detection
```

In this example, the results show that there are three tags set, with a combined value of 0x40001400.

- Enable pool tagging (ptg) = 0x400
- Create user mode stack trace database (ust) = 0x1000
- Enable bad handles detection (bhd) = 0x40000000

The following command displays the flags set for the current session. It uses the **/k** parameter to indicate kernel mode.

```
gflags /k
```

The following command displays flags set in the registry for the image file notepad.exe. It uses the **/i** parameter to indicate image file mode and specifies the image file.

```
gflags /i notepad.exe
```

Remember that the flag value displayed might not be the current, effective flag value. Changes to the system-wide flags are not effective until you restart Windows. Changes to image file flag settings are not effective until you restart the program.

Example 2: Setting a Flag by Using a Flag Abbreviation

12/5/2018 • 2 minutes to read • Edit Online

The following command sets the Show loader snaps flag for the notepad.exe image file. **Show Loader Snaps** takes snapshots of the load process, capturing in detail the loading and unloading of executable images and their supporting library modules.

The command uses the **/i** parameter to indicate image file mode and specifies the name of the image file notepad.exe. To identify the flag, the command uses **sls**, the abbreviation for **Show Loader Snaps**, and it precedes the abbreviation with a plus sign (+) to indicate that the flag is set. Without the plus sign, the command has no effect.

```
gflags /i notepad.exe +sls
```

In response, GFlags displays the flags set for notepad.exe. The display indicates that the command is successful. The **Show Loader Snaps** feature is enabled for all new sessions of the Notepad program.

```
Current Registry Settings for notepad.exe executable are: 00000002
    sls - Show Loader Snaps
```

Example 3: Setting a Flag by Using Its Hexadecimal Value

12/5/2018 • 2 minutes to read • Edit Online

The following command sets the system-wide Enable page heap flag. **Enable Page Heap** adds a guard page and other tracking features to each heap allocation.

The command uses the **/r** parameter to indicate system-wide registry mode. To identify the flag, the command uses **2000000**, which represents 0x2000000, the hexadecimal value for **Enable page heap**.

Although the command sets a flag, it omits the plus (+) sign. When using hexadecimal values, the sign is optional and add (+) is the default.

```
gflags /r 2000000
```

In response, GFlags displays the system-wide flags set in the registry. The display indicates that the command is successful. The **Enable page heap** feature will be enabled for all processes when Windows restarts.

```
Current Boot Registry Settings are: 02000000
    hpa - Enable page heap
```

Example 4: Setting Multiple Flags

12/5/2018 • 2 minutes to read • Edit Online

The following command sets these three flags for the current session:

- Enable heap free checking (hfc, 0x20)
- Enable heap parameter checking (hpc, 0x40)
- Enable heap validation on call (hvc, 0x80)

This command uses the **/k** parameter to specify kernel mode (session only). It sets the value for kernel mode to **E0** (0xE0), the sum of the hexadecimal values of the selected flags (0x20 + 0x40 + 0x80).

```
gflags /k e0
```

In response, GFlags displays the revised value of flags set for the session. The display indicates that the command is successful and that the three flags specified in the command are set.

```
Current Running Kernel Settings are: 000000e0
    hfc - Enable heap free checking
    hpc - Enable heap parameter checking
    hvc - Enable heap validation on call
```

You can use several different GFlags commands to set flags. Each of the following commands has the same effect as the command used in this example and the methods can be used interchangeably:

```
gflags /k +20 +40 +80
gflags /k +E0
gflags /k +hfc +hpc +hvc
```

Kernel (run time) flags are effective immediately and remain effective until Windows shuts down.

Example 5: Clearing a Flag

12/5/2018 • 2 minutes to read • Edit Online

The following command clears the system-wide Enable page heap flag set in the registry. The command uses the **/r** parameter to indicate the system-wide registry mode and **hpa**, the abbreviation for the **Enable page heap** flag. The minus sign (-) indicates that the flag is to be cleared.

```
gflags /r -hpa
```

In response, GFlags displays the current value of the system-wide registry entry. The display indicates that the command is successful and that there are no longer any flags set.

```
Current Boot Registry Settings are: 00000000
```

Note that the following command, which uses the hexadecimal value of the **Enable Page Heap** flag, has the same effect as the command used in this example. These commands can be used interchangeably:

```
gflags /r -02000000
```

Example 6: Clearing All Flags

12/5/2018 • 2 minutes to read • Edit Online

This example demonstrates two different ways to clear all flags set in the registry and for the session:

- Subtract the current flag value.
- Subtract high values.

Note The methods demonstrated by this example clear flags only. They do not reset the maximum stack trace size or kernel special pool tag to the default values.

Subtract the Current Flag Value

The following command clears all flags set in the system-wide flag entry in the registry by subtracting the current value of the entry. In this example, the current value is 0xE0. The command uses the **/r** parameter to indicate the system-wide registry mode and the E0 value with a minus sign (-) to subtract E0 from the flag value.

```
gflags /r -E0
```

In response, GFlags displays the revised value of system-wide flag registry entry. A value of zero indicates that the command is successful and that there are no longer any system-wide flags set in the registry.

```
Current Boot Registry Settings are: 00000000
```

Note that the following commands have the same effect as the command used in this example and can be used interchangeably:

```
gflags /r -20 -40 -80
gflags /r -hfc -hpc -hvc
```

Subtract High Values

The following command clears all system-wide flags by subtracting high values (0xFFFFFFFF) from the system-wide flag setting.

```
gflags /r -ffffffff
```

In response, GFlags displays the revised value of the system-wide flag entry. A value of zero indicates that the command is successful and that there are no longer any system-wide flags set in the registry.

```
Current Boot Registry Settings are: 00000000
```

Tip Type this command into Notepad, then save the file as clearflag.bat. Thereafter, to clear all flags, just type **ClearFlag**.

Finally, the following example demonstrates that the intuitive method of clearing all flags does not work.

The following command appears to set the value of the system-wide flag entry to 0. However, it actually adds zero to the system-wide flag value. In this example, the current value of the system-wide flag entry is 0xE0.

```
gflags /r 0
```

In response, GFlags displays the system-wide flag value after the command completes:

```
Current Boot Registry Settings are: 000000e0
    hfc - Enable heap free checking
    hpc - Enable heap parameter checking
    hvc - Enable heap validation on call
```

The command has no effect because it adds the value 0 to system-wide flag entry.

Example 7: Clearing All Flags for an Image File

12/5/2018 • 2 minutes to read • Edit Online

The following command clears all flags and image debugger options for an image file. The command adds high-values (0xFFFFFFFF) to the current flag value. GFlags responds by deleting the **GlobalFlag** registry entry for the image file, thereby deleting all of the values it stores.

This command does not affect flags set in the system-wide **GlobalFlag** registry entry or flags set for the session (kernel mode).

```
gflags /i notepad.exe ffffffff
```

In response, GFlags displays a message indicating that there are no flags set for the image file:

```
No Registry Settings for notepad.exe executable
```

Example 8: Enlarging the User-Mode Stack Trace Database

12/5/2018 • 2 minutes to read • Edit Online

The following GFlags command increases the maximum size of the user-mode stack trace database for myapp.exe, a fictitious program, from 8 MB to 24 MB.

The command uses the **/i** parameter to specify the image file. It uses the **/tracedb** parameter to set the maximum stack trace database size and the value 24 to indicate the size in megabytes. The command uses decimal units. (Hexadecimal units are not valid.)

```
gflags /i MyApp.exe /tracedb 24
```

As the following error message indicates, this command fails because the Create user mode stack trace database (+ust) flag is not set for the MyApp image file. You cannot set the size of a trace database until you create one.

```
Failed to set the trace database size for `MyApp.exe'
```

The following commands fix the error. The first command creates a trace database for myapp.exe and the second command sets the maximum size of the trace database to 24 MB. These commands cannot be combined into a single command. The following display shows the commands and the success message from GFlags.

```
gflags /i MyApp.exe +ust

Current Registry Settings for MyApp.exe executable are: 00001000
    ust - Create user mode stack trace database

gflags /i MyApp.exe /tracedb 24

Trace database size for `MyApp.exe' set to 24 Mb.
```

GFlags can change the size of the user-mode stack trace database, but it does not display it. To display the trace database size, use registry APIs, Regedit, or Reg (reg.exe), a tool included in Windows XP and Windows Server 2003, to check the value of the **StackTraceDatabaseSizeInMB** registry entry (HKLM\Software\Microsoft\Windows NT\CurrentVersion\Image File Execution Options*ImageFileName***StackTraceDatabaseSizeInMB**).

(A version of Reg is included in Windows XP, but that version does not permit the **/v** and **/s** switches in the same command.)

The following command uses Reg to query the value of **StackTraceDatabaseSizeInMB**:

```
reg query "HKLM\Software\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\MyApp.exe" /v
StackTraceDatabaseSizeInMB
```

In response, Reg displays the value of **StackTraceDatabaseSizeInMB**, which confirms that the new value, 24 (0x18), was set. This value becomes effective when you restart myapp.exe.

```
! REG.EXE VERSION 3.0

HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\MyApp.exe
    StackTraceDatabaseSizeInMB  REG_DWORD       0x18
```

Tip Type the **reg query** command into Notepad, then save the file as tracedb.bat. Thereafter, to display the value of **StackTraceDatabaseSizeInMB**, just type **TraceDb**.

Example 9: Detecting a Pool Memory Leak

12/5/2018 • 2 minutes to read • Edit Online

The following example uses GFlags to set the system-wide Enable pool tagging flag in the registry. Then, it uses PoolMon (poolmon.exe), a tool in the Windows Driver Kit, to display the size of the memory pools.

PoolMon monitors the bytes in the paged and nonpaged memory pools and sorts them by pool tag. By running PoolMon periodically, you can identify pools that expand continuously over time. This pattern often indicates a memory leak.

Note Pool tagging is permanently enabled in Windows Server 2003 and later versions of Windows. On these systems, the **Enable pool tagging** check box on the **Global Flags** dialog box is dimmed, and commands to enable or disable pool tagging fail. If pool tagging is not enabled, PoolMon fails and displays the following message: "Query pooltags Failed c0000002."

To detect a pool memory leak

1. To enable pool tagging for all processes in versions of Windows earlier than Windows Server 2003, set the system-wide Enable pool tagging flag in the registry. The following command line uses the flag abbreviation method, but you can identify the flag by its hexadecimal value or use the **Global Flags** dialog box:

   ```
   gflags /r +ptg
   ```

2. Restart the computer to make the registry change effective.

3. Run PoolMon periodically by using the following command. In this command, the **/b** parameter sorts the pools in descending size order.

   ```
   poolmon /b
   ```

   In response, PoolMon displays allocations from the memory pools in descending size order , including the number of allocate operations and free operations, and the amount of memory remaining in the pool (in the Bytes column).

   ```
   Memory: 16224K Avail: 4564K PageFlts: 31 InRam Krnl: 684K P: 680K
    Commit: 24140K Limit: 24952K Peak: 24932K  Pool N: 744K P: 2180K

    Tag  Type    Allocs          Frees         Diff   Bytes      Per Alloc
   ------------------------------------------------------------------------

    CM   Paged    1283 (   0)    1002 (   0)    281 1377312 (     0) 4901
   Strg  Paged   10385 (  10)    6658 (   4)   3727  317952 (   512)   85
    Fat  Paged    6662 (   8)    4971 (   6)   1691  174560 (   128)  103
   MmSt  Paged     614 (   0)     441 (   0)    173   83456 (     0)  482
   ```

   If the value in the **Bytes** column for an allocation expands continuously for no obvious reason, there might be a memory leak in that pool.

4. Clear the **Enable pool tagging** flag.

   The following command line uses the flag abbreviation method, but you can identify the flag by its hexadecimal value or use the **Global Flags** dialog box:

```
gflags /r -ptg
```

5. Restart Windows to make the registry change effective.

Note Use the append symbol (**>>**) to redirect the PoolMon output to a log file. Later, you can examine the log file for pool size trends. For example:

```
poolmon.exe /b >> poolmon.log
```

Example 10: Detecting a Heap Memory Leak in a Process

12/5/2018 • 2 minutes to read • Edit Online

This example uses GFlags and User Mode Dump Heap (UMDH, umdh.exe), a tool included in Microsoft Debugging Tools for Windows.

To detect a leak in heap memory in notepad.exe

1. Set the Create user mode stack trace database (**ust**) flag for the notepad.exe image file.

   The following command uses GFlags to set the **Create user mode stack trace database** flag. It uses the **/i** parameter to identify the image file and the **ust** abbreviation for the flag.

   ```
   gflags /i Notepad.exe +ust
   ```

   As a result of this command, a user-mode stack trace is created for all new instances of the Notepad process.

2. Set the symbol file path.

   The following command creates an environment variable that stores the path to the directory of symbol files:

   ```
   set _NT_SYMBOL_PATH=C:\Windows\symbols
   ```

3. Start Notepad.

4. Find the process identifier (PID) of the Notepad process.

   You can find the PID of any running process from Task Manager or Tasklist (tasklist.exe), a tool included in Windows XP Professional and Windows Server 2003 operating systems. In this example, the Notepad PID is 1228.

5. Run UMDH.

   The following command runs UMDH (umdh.exe). It uses the **-p:** parameter to specify the PID that, in this example, is 1228. It uses the **/f:** parameter to specify the name and location of the output file for the heap dump, notepad.dmp.

   ```
   umdh -p:1228 -f:notepad.dmp
   ```

   In response, UMDH writes a complete dump of all active heaps to the notepad.dmp file.

Example 11: Enabling Page Heap Verification

12/5/2018 • 2 minutes to read • Edit Online

The following commands enable full and standard page heap verification for myapp.exe, a fictitious program.

The first command enables *standard* page heap verification for myapp.exe. It uses the **/p** parameter to enable page heap for a process. By default, **/p** enables standard page heap.

```
gflags /p /enable myapp.exe
```

The following commands enable *full* page heap verification for the myapp.exe program. Although these commands create different settings in the registry, they are all functionally equivalent to selecting the **Enable page heap** check box for the myapp.exe image file in the **Global Flags** dialog box. These methods can be used interchangeably.

```
gflags /p /enable myapp.exe /full
gflags /i myapp.exe +hpa
gflags /i myapp.exe +02000000
```

The following commands disable full or standard page heap verification for the myapp.exe program, regardless of the command or dialog box method used to enable page heap verification.

```
gflags /p /disable myapp.exe
gflags /i myapp.exe -hpa
gflags /i myapp.exe -02000000
```

Note When using the **/debug** or **/kdebug** parameters, use the **/p /disable** parameters to turn off the page heap verification (not the **/i -hpa** parameters). The **/p /disable** parameters disable page heap verification and delete registry entries that the debugger reads.

Example 12: Using Page Heap Verification to Find a Bug

12/5/2018 • 4 minutes to read • Edit Online

The following series of commands demonstrates how to use the page heap verification features of GFlags and the NTSD debugger to detect an error in heap memory use. In this example, the programmer suspects that a fictitious application, pheap-buggy.exe, has a heap error, and proceeds through a series of tests to identify the error.

For details on NTSD, see Debugging Using CDB and NTSD.

Step 1: Enable standard page heap verification

The following command enables standard page heap verification for pheap-buggy.exe:

```
gflags /p /enable pheap-buggy.exe
```

Step 2: Verify that page heap is enabled

The following command lists the image files for which page heap verification is enabled:

```
gflags /p
```

In response, GFlags displays the following list of programs. In this display, **traces** indicates standard page heap verification, and **full traces** indicates full page heap verification. In this case, pheap-buggy.exe is listed with **traces**, indicating that standard page heap verification is enabled, as intended.

```
pheap-buggy.exe: page heap enabled with flags (traces )
```

Step 3: Run the debugger

The following command runs the **CorruptAfterEnd** function of pheap-buggy.exe in NTSD with the **-g** (ignore initial breakpoint) and **-x** (set second-chance break on access violation exceptions) parameters:

```
ntsd -g -x pheap-buggy CorruptAfterEnd
```

When the application fails, NTSD generates the following display, which indicates that it detected an error in pheap-buggy.exe:

```
============================================================
VERIFIER STOP 00000008: pid 0xAA0: corrupted suffix pattern

        00C81000 : Heap handle
        00D81EB0 : Heap block
        00000100 : Block size
#         00000000 :
============================================================

Break instruction exception - code 80000003 (first chance)
eax=00000000 ebx=00d81eb0 ecx=77f7e257 edx=0006fa18 esi=00000008 edi=00c81000
eip=77f7e098 esp=0006fc48 ebp=0006fc5c iopl=0         nv up ei pl zr na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
ntdll!DbgBreakPoint:
77f7e098 cc               int     3
```

The header information includes the address of the heap with the corrupted block (00C81000 : Heap handle), the address of the corrupted block (00D81EB0 : Heap block), and the size of the allocation (00000100 : Block size).

The "corrupted suffix pattern" message indicates that the application violated the data integrity pattern that GFlags inserted after the end of the pheap-buggy.exe heap allocation.

Step 4: Display the call stack

In the next step, use the addresses that NTSD reported to locate the function that caused the error. The next two commands turn on line number dumping in the debugger and display the call stack with line numbers.

```
C:\>.lines

Line number information will be loaded

C:\>kb

ChildEBP RetAddr  Args to Child
WARNING: Stack unwind information not available. Following frames may be wrong.
0006fc5c 77f9e6dd 00000008 77f9e3e8 00c81000 ntdll!DbgBreakPoint
0006fcd8 77f9f3c8 00c81000 00000004 00d81eb0 ntdll!RtlpNtEnumerateSubKey+0x2879
0006fcfc 77f9f5bb 00c81000 01001002 00000010 ntdll!RtlpNtEnumerateSubKey+0x3564
0006fd4c 77fa261e 00c80000 01001002 00d81eb0 ntdll!RtlpNtEnumerateSubKey+0x3757
0006fdc0 77fc0dc2 00c80000 01001002 00d81eb0 ntdll!RtlpNtEnumerateSubKey+0x67ba
0006fe78 77fbd87b 00c80000 01001002 00d81eb0 ntdll!RtlSizeHeap+0x16a8
0006ff24 010013a4 00c80000 01001002 00d81eb0 ntdll!RtlFreeHeap+0x69
0006ff3c 01001450 00000000 00000001 0006ffc0 pheap-buggy!TestCorruptAfterEnd+0x2b
[d:\nttest\base\testsrc\kernel\rtl\pageheap\pheap-buggy.cxx @ 185]
0006ff4c 0100157f 00000002 00c65a68 00c631d8 pheap-buggy!main+0xa9
[d:\nttest\base\testsrc\kernel\rtl\pageheap\pheap-buggy.cxx @ 69]
0006ffc0 77de43fe 00000000 00000001 7ffdf000 pheap-buggy!mainCRTStartup+0xe3 [crtexe.c @ 349]
0006fff0 00000000 0100149c 00000000 78746341 kernel32!DosPathToSessionPathA+0x204
```

As a result, the debugger displays the call stack for pheap-buggy.exe with line numbers. The call stack display shows that the error occurred when the **TestCorruptAfterEnd** function in pheap-buggy.exe tried to free an allocation at 0x00c80000 by calling **HeapFree**, a redirect to **RtlFreeHeap**.

The most likely cause of this error is that the program wrote past the end of the buffer that it allocated in this function.

Step 5: Enable full page heap verification

Unlike standard page heap verification, full page heap verification can catch the misuse of this heap buffer as soon as it occurs. The following command enables full page heap verification for pheap-buggy.exe:

```
gflags /p /enable pheap-buggy.exe /full
```

Step 6: Verify that full page heap is enabled

The following command lists the programs for which page heap verification is enabled:

```
gflags /p
```

In response, GFlags displays the following list of programs. In this display, **traces** indicates standard page heap verification, and **full traces** indicates full page heap verification. In this case, pheap-buggy.exe is listed with **full traces**, indicating that full page heap verification is enabled, as intended.

```
pheap-buggy.exe: page heap enabled with flags (full traces )
```

Step 7: Run the debugger again

The following command runs the **CorruptAfterEnd** function of pheap-buggy.exe in the NTSD debugger with the **-g** (ignore initial breakpoint) and **-x** (set second-chance break on access violation exceptions) parameters:

```
ntsd -g -x pheap-buggy CorruptAfterEnd
```

When the application fails, NTSD generates the following display, which indicates that it detected an error in pheap-buggy.exe:

```
Page heap: process 0x5BC created heap @ 00880000 (00980000, flags 0x3)
ModLoad: 77db0000 77e8c000   kernel32.dll
ModLoad: 78000000 78046000   MSVCRT.dll
Page heap: process 0x5BC created heap @ 00B60000 (00C60000, flags 0x3)
Page heap: process 0x5BC created heap @ 00C80000 (00D80000, flags 0x3)
Access violation - code c0000005 (first chance)
Access violation - code c0000005 (!!! second chance !!!)
eax=00c86f00 ebx=00000000 ecx=77fbd80f edx=00c85000 esi=00c80000 edi=00c16fd0
eip=01001398 esp=0006ff2c ebp=0006ff4c iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000206
pheap-buggy!TestCorruptAfterEnd+1f:
01001398 889801010000     mov     [eax+0x101],bl          ds:0023:00c87001=??
```

With full page heap verification enabled, the debugger breaks at an access violation. To find the precise location of the access violation, turn on line number dumping and display the call stack trace.

The numbered call stack trace appears as follows: The line displaying the problem appears in bold text.

```
ChildEBP RetAddr  Args to Child
0006ff3c 01001450 00000000 00000001 0006ffc0 pheap-buggy!TestCorruptAfterEnd+0x1f
[d:\nttest\base\testsrc\kernel\rtl\pageheap\pheap-buggy.cxx @ 184]
0006ff4c 0100157f 00000002 00c16fd0 00b70eb0 pheap-buggy!main+0xa9
[d:\nttest\base\testsrc\kernel\rtl\pageheap\pheap-buggy.cxx @ 69]
0006ffc0 77de43fe 00000000 00000001 7ffdf000 pheap-buggy!mainCRTStartup+0xe3 [crtexe.c @ 349]
WARNING: Stack unwind information not available. Following frames may be wrong.
0006fff0 00000000 0100149c 00000000 78746341 kernel32!DosPathToSessionPathA+0x204
```

The stack trace shows that the problem occurs in line 184 of pheap-buggy.exe. Because full page heap verification is enabled, the call stack starts in the program code, not in a system DLL. As a result, the violation was caught where it happened, instead of when the heap block was freed.

Step 8: Locate the error in the code

A quick inspection reveals the cause of the problem: The program tries to write to the 257th byte (0x101) of a 256-byte (0x100) buffer, a common off-by-one error.

```
*((PCHAR)Block + 0x100) = 0;
```

Example 13: Listing Image Files with Global Flags

12/5/2018 • 2 minutes to read • Edit Online

GFlags displays the flags set for a particular image file, but it does not display all image files that have flags set.

Windows stores flags for an image file that the **GlobalFlag** registry entry in a registry subkey named for the image file in the following registry path, **HKEY_LOCAL_MACHINE\ SOFTWARE\ Microsoft\ Windows NT\ CurrentVersion\ Image File Execution Options*ImageFileName*\GlobalFlag**.

To determine which image files have flags set, use Reg (reg.exe), a tool included in Windows Server 2003.

The following Reg **Query** command searches for the **GlobalFlag** registry entry in the specified registry path. The **-v** parameter specifies the **GlobalFlag** registry entry. The **/s** parameter makes the search recursive.

```
reg query "HKLM\Software\Microsoft\Windows NT\CurrentVersion\Image File Execution Options" /v GlobalFlag /s
```

In response, Reg displays all instances of the **GlobalFlag** registry entry in the path and the value of the entry.

```
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\notepad.exe
    GlobalFlag    REG_SZ    0x00001000

HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\photohse.EXE
    GlobalFlag    REG_SZ    0x00200000

HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\printhse.EXE
    GlobalFlag    REG_SZ    0x00200000
```

Tip Type the **Reg** command into Notepad, then save the file as imageflags.bat. Thereafter, to find image files for which flags have been set, just type **ImageFlags**.

Example 14: Configuring Special Pool

12/5/2018 • 5 minutes to read • Edit Online

Beginning in Windows Vista, you can configure the Special Pool feature as a kernel flag setting or as a registry setting. If you configure it as a kernel flag (run time) setting, you do not need to restart the computer to make the change effective. In earlier versions of Windows, Special Pool is available only as a registry setting.

Also, beginning in Windows Vista, you can set and configure the Special Pool feature from the command line. In earlier versions of Windows, you can set and configure the Special Pool feature only in the Global Flags dialog box.

Request Special Pool by pool tag without rebooting

The following command requests special pool for all allocations with the **Tag1** pool tag. This setting becomes effective immediately, but it is lost if you shut down or restart Windows.

This command uses the **/k** parameter to specify a kernel flag (run time) setting and the +spp abbreviation to set a special pool request.

```
gflags /k +spp Tag1
```

Gflags responds by printing:

```
Special Pool set to 0x31676154
PoolTagOverruns set to 0x1
Current Running Kernel Settings are: 00000000
```

Notice that the special pool allocation request is not a kernel flag setting and is not reflected in the kernel settings value.

Also, a special pool allocation request does not change the value of the overrun (0x1) or underrun (0x0) setting for special pool. To change from overruns, the default, to underruns, use the Gflags Dialog Box. For information, see Detecting Overruns and Underruns.

You cannot display the pool tag at the command line. To verify that the pool tag is a kernel setting, use the Gflags Dialog Box.

Request Special Pool by pool tag in the registry

The following command requests special pool for all allocations with the **Tag1** pool tag. Because this setting is stored in the registry, you must restart the computer to make it effective, but it remains effective until you change it.

This command uses the **/r** parameter to specify a registry setting and the +spp abbreviation to set a special pool request.

```
gflags /r +spp Tag1
```

Gflags responds by printing:

```
Special Pool set to 0x31676154
PoolTagOverruns set to 0x1
Current Boot Registry Settings are: 00000000
```

Notice that the special pool allocation request is not a registry flag setting and is not reflected in the registry settings value.

Also, a special pool allocation request does not change the value of the overrun (0x1) or underrun (0x0) setting for special pool. To change from overruns, the default, to underruns, use the Gflags Dialog Box. For information, see Detecting Overruns and Underruns.

To verify that the value has been added to the registry, use Reg or Regedit to display the value of the **PoolTag** entry in the **HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management** key.

For example:

```
c:>reg query "HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management" -v PoolTag
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management
    PoolTag    REG_DWORD    0x31676154
```

Request Special Pool by size without rebooting

The following command requests special pool for allocations of 1 to 8 bytes on an x86 computer with a PAGE_SIZE of 0x1000 and allocation granularity of 8 bytes.

This command uses the **/k** parameter to specify a kernel flag (run time) setting and the +spp abbreviation to set a special pool request. The size value is preceded by 0x to indicate that it is a size and not a pool tag.

The value, 0x10, is calculated by adding the allocation granularity (8 bytes) to the largest size in the range (8 bytes) for a total of 16 bytes (0x10). For help in determining the correct value to enter, see "Selecting an Allocation Size" in Special Pool.

```
gflags /k +spp 0x10
```

Gflags responds by printing:

```
Special Pool set to 0x10
PoolTagOverruns set to 0x1
Current Running Kernel Settings are: 00000000
```

Again, the special pool allocation request is not a kernel flag setting and is not reflected in the kernel settings value.

Also, a special pool allocation request does not change the value of the overrun (0x1) or underrun (0x0) setting for special pool. To change from overruns, the default, to underruns, use the Gflags Dialog Box. For information, see Detecting Overruns and Underruns.

Request Special Pool by size in the registry

The following command requests special pool for allocations of 1024 to 1040 bytes on an x64 computer with a PAGE_SIZE of 0x1000 and allocation granularity of 16 bytes.

This command uses the **/r** parameter to specify a system-wide registry setting and the +spp abbreviation to set a special pool request. The size value is preceded by 0x to indicate that it is a size and not a pool tag.

The value, 0x420, is calculated by adding the allocation granularity (16 bytes) to the largest size in the range (1040

bytes) for a total of 1056 bytes (0x420). For help in determining the correct value to enter, see "Selecting an Allocation Size" in Special Pool.

```
gflags /r +spp 0x420
```

Gflags responds by printing:

```
Special Pool set to 0x420
PoolTagOverruns set to 0x1
Current Boot Registry Settings are: 00000000
```

Again, the special pool allocation request is not a registry flag setting and is not reflected in the registry settings value.

Also, a special pool allocation request does not change the value of the overrun (0x1) or underrun (0x0) setting for special pool. To change from overruns, the default, to underruns, use the Gflags Dialog Box. For information, see Detecting Overruns and Underruns.

To verify that the value has been added to the registry, use Reg or Regedit to display the value of the **PoolTag** entry in the **HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management** key.

For example:

```
c:>reg query "HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management" -v PoolTag
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management
    PoolTag    REG_DWORD    0x420
```

Cancel a Special Pool Request

The following command cancels a request for Special Pool as a kernel flag (run time) setting. The command is the same for a request by pool tag or by size.

```
gflags /k -spp
```

The following command cancels a request for Special Pool as a registry setting. The command is the same for a request by pool tag or by size.

```
gflags /r -spp
```

When the command is successful, Gflags responds by printing:

```
Special Pool value has been deleted.
```

Example 15: Using Object Reference Tracing

12/5/2018 • 5 minutes to read • Edit Online

Object Reference Tracing is a Windows feature that records a sequential stack trace when an object is referenced or dereferenced. It is designed to detect errors in object handling that can lead to crashes or memory leaks. Some of these errors are difficult to detect because they do not appear consistently. For detailed information, see Object Reference Tracing.

You can configure Object Reference Tracing by using the **Global Flags** dialog box or at a command prompt. The following examples use the command prompt. For information about using the **Global Flags** dialog box to configure Object Reference Tracing, see Configuring Object Reference Tracing.

You can use Gflags to enable, disable, and configure Object Reference Tracing. The process is as follows:

- **Use Gflags to enable Object Reference Tracing** in the registry or as a kernel flag (run time) setting. If you add the setting to the registry, you must restart the computer to start tracing. If you enable the run time version of the settings, the trace starts immediately, but the trace settings revert to those in the registry key when you shut down or restart the computer.
- **Start the process that creates the suspect object**. The trace includes only objects created by processes that are started after the trace begins. If the process starts during or soon after restarting, add the trace settings to the registry, and then restart the system.
- **Use the !obtrace debugger extension** to view the trace. By default, the trace is maintained until the object is destroyed, but you can use the **/p** parameter to maintain the trace until tracing is disabled.
- **Use Gflags to disable Object Reference Tracing**.in the registry or as a kernel flag (run time) setting. If you delete the setting from the registry, you must restart the computer to end tracing. If you disable the run time version of the settings, the trace ends immediately, but the trace settings revert to those in the registry when you shut down or restart the computer.

These examples show how to use Gflags to enable and disable object reference tracing. \

Enable Run-time Tracing

The following command enables Object Reference Tracing at the command prompt. The command uses the **/ko** parameter to enable Object Reference Tracing as a kernel flag (run time) setting. The command uses the **/t** parameter to specify the pool tags **Tag1** and **Fred**. As a result, all objects that are created with **Tag1** or **Fred** are traced.

```
gflags /ko /t Tag1;Fred
```

Because the command changes the kernel flag (run-time) settings, object reference tracing starts immediately. The trace will include all objects with the pool tags **Tag1** or **Fred** that are created by processes that start after the command is submitted.

Gflags responds by printing the following message:

```
Running Kernel Settings :
Object Ref Tracing Enabled
        Temporary Traces
        Pool Tags: Tag1;Fred
        Process Name: All Processes
```

This message indicates that Object Reference Tracing is enabled. "Temporary Traces" indicates that all records of the trace are deleted when the object is destroyed. To make the trace "permanent," use the **/p** parameter, which directs Windows to retain the trace data until Object Reference Tracing is disabled, or the computer is shut down or restarted.

Enable Tracing in the Registry

The following command adds an Object Reference Tracing configuration to the registry. The trace that you configure begins when you restart the computer.

The command uses the **/ro** parameter to enable Object Reference Tracing as a registry setting. The command uses the **/i** to specify the process for notepad.exe and the **/t** parameter to specify the pool tags **Tag1** and **Fred**. As a result, all objects that are created by the Notepad process with the **Tag1** or **Fred** pool tags are traced. The command also uses the **/p** parameter, which retains the trace data until the tracing is disabled.

```
gflags /ro /t Tag1;Fred /i Notepad.exe /p
```

When you submit the command, Gflags stores the information in the registry. However, because registry settings are not effective until you restart the computer, this object reference tracing is configured, but is not yet started.

Gflags responds by printing the following message:

```
Boot Registry Settings :
Object Ref Tracing Enabled
        Permanent Traces
        Pool Tags: Tag1;Fred
        Process Name: Notepad.exe
```

The message indicates that Object Reference Tracing is enabled in the registry. "Permanent Traces" indicates that the trace data will be retained until you shut down or restart the computer. The message also lists the pool tags and image file names that will be traced.

Display the Object Reference Tracing Configuration

You can display the Object Reference Tracing configuration that is currently effective or is stored in the registry to be used when the computer is restarted.

In this example, there is one Object Reference Tracing configuration stored in the registry and a different one configured for run time. The run-time trace begins immediately (and overrides any registry settings). However, if you restart the system, the run-time settings are lost, and the Object Reference Tracing session registry settings become effective.

The following command displays the run time Object Reference Tracing configuration. It uses the **/ko** parameter with no other parameters.

```
gflags /ko
```

```
Running Kernel Settings :
Object Ref Tracing Enabled
        Temporary Traces
        Pool Tags: Tag1;Fred
        Process Name: All Processes
```

If Object Reference Tracing is enabled, as it is in this example, the settings that are displayed describe a trace that is in progress.

The following command displays the Object Reference Tracing configuration data stored in the registry. It uses the **/ro** parameter with no other parameters.

```
gflags /ro
```

In response, Gflags displays the data stored in the registry:

```
Boot Registry Settings :
Object Ref Tracing Enabled
        Permanent Traces
        Pool Tags: Tag1;Fred
        Process Name: Notepad.exe
```

If you have restarted the computer since you added the Object Reference Tracing configuration to the registry, the settings that are displayed in response to a gflags /ro command describe the trace that is in progress. However, if you have not yet restarted, or you have restarted, but then started a run-time object reference trace (**/ko**), the settings that are stored in the registry are not currently effective, but they will become effective again when you reboot the system.

Disable Object Reference Tracing

When you disable run-time (kernel flag) Object Reference Tracing settings, the trace stops immediately. When you disable Object Reference Tracing settings in the registry, the trace stops when you restart the computer.

The following command disables run-time Object Reference Tracing. It uses the **/d** parameter to disable all settings. You cannot disable settings selectively.

```
gflags /ko -d
```

When the command succeeds, Gflags responds with the following message:

```
Running Kernel Settings :
Object Ref Tracing Disabled
```

The following command disables run-time Object Reference Tracing.

The following command disables Object Reference Tracing settings in the registry. It uses the **/d** parameter to disable all settings. You cannot disable settings selectively. This command is effective when you restart the computer.

```
gflags /ro -d
```

When the command succeeds, Gflags responds with the following message:

```
Boot Registry Settings :
Object Ref Tracing Disabled
```

Kill Tool

10/8/2019 • 2 minutes to read • Edit Online

The Kill tool, kill.exe, terminates one or more processes and all of their threads. This tool works only on processes running on the local computer.

Where to get Kill Tool

Kill.exe is included in Debugging Tools for Windows.

Kill Tool command-line options

```
kill [/f] { PID | Pattern* }
```

Parameters

/f Forces the termination of the process without prompting the user for confirmation. This option is required to terminate a protected process, such as a system service.

PID Specifies the process identifier (PID) of the task to be terminated.

To find the PID for a task, use TaskList in Microsoft Windows XP and later or TList in Windows 2000.

*Pattern**
Specifies all or part of the name of a task or window. The Kill tool terminates all processes whose process names or window names match the pattern. The asterisk is required.

Before using a pattern that might match many process or window names unintentionally, use the **tlist** *pattern* command to test the pattern. See TList for details.

Examples

The following command terminates processes whose names begin with "myapp."

```
kill myapp*
```

The following command terminates the process whose process ID (PID) is 2520:

```
kill 2520
```

The following command terminates processes whose names begin with "my*." It does not prompt for confirmation. This command succeeds even when this process is a system service:

```
kill /f my*
```

Related topics

Tools Included in Debugging Tools for Windows

Logger and LogViewer

12/5/2018 • 2 minutes to read • Edit Online

The *Logger* and *LogViewer* tools provide an alternative to standard user-mode debugging. Logger can monitor the actions of a user-mode target application and record all of its API calls. The resulting information can be displayed in the debugger, saved as a text file, or displayed in a powerful interactive format by the LogViewer tool..

Where to get Logger and LogViewer

Logger and LogViewer are included in Debugging Tools for Windows.

In this section

Logger

LogViewer

The Logger Manifest

Related topics

Tools Included in Debugging Tools for Windows

Logger

12/5/2018 • 2 minutes to read • Edit Online

The Logger tool can be activated through two different vehicles. One way is to use the stand-alone Logger.exe program. The other is to start CDB or WinDbg, and use the Logexts.dll debugger extensions. Both of these methods will produce the same type of log output. However, the best vehicle to use on any NT-based operating system is CDB or WinDbg with the Logexts.dll extension commands.

The Logger vehicle works as well, but using the debugger gives you the full power of the debugger along with the power of Logger.

This section includes:

Using the Debugger and Logexts.dll

Using Logger.exe

Logger Restrictions and Limitations

Using the Debugger and Logexts.dll

12/5/2018 • 3 minutes to read • Edit Online

One way to activate Logger is to start CDB or WinDbg and attach to a user-mode target application as usual. Then, use the **!logexts.logi** or **!logexts.loge** extension command.

This will insert code at the current breakpoint that will jump off to a routine that loads and initializes Logexts.dll in the target application process. This is referred to as "injecting Logger into the target application."

There will actually be two instances of Logexts.dll running, since this module is both a debugger extension DLL and the program that is injected into the target application. The debugger and target instances of Logexts.dll communicate through a shared section of memory that includes the output file handles, current category mask, and a pointer to the log output buffer.

Attaching to the Target Application

For information about attaching the debugger to the target application, see Debugging a User-Mode Process Using WinDbg or Debugging a User-Mode Process Using CDB.

Using the Logger Extension Commands

For the full syntax of each extension, see its reference page.

!logexts.logi

Injects Logger into the target application. This initializes logging, but does not enable it.

!logexts.loge

Enables logging. If **!logexts.logi** has not been used, this extension will initialize and then enable logging.

!logexts.logd

Disables logging. This will cause all API hooks to be removed in an effort to allow the program to run freely. COM hooks are not removed because they cannot be re-enabled at will.

!logexts.logo

Displays or modifies output options. Three types of output are possible: messages sent directly to the debugger, a text file, or an .lgv file. The .lgv file contains much more information than the other two; it can be read with LogViewer.

If you disable the text file output, a .txt file of size zero will still be created. This may overwrite a previously-saved text file in the same location.

!logexts.logc

Displays available API categories, controls which categories will be logged and which will not, and displays the APIs that are contained in any category.

If a category is disabled, the hooks for all APIs in that category will be removed so that there is no longer any performance overhead. COM hooks are not removed because they cannot be re-enabled at will.

Enabling only certain categories can be useful when you are only interested in a particular type of interaction that the program is having with Windows -- for example, file operations. This reduces the log file size and also reduces the effect that Logger has on the execution speed of the process.

!logexts.logb

Displays or flushes the current output buffer. As a performance consideration, log output is flushed to disk only when the output buffer is full. By default, the buffer is 2144 bytes.

Since the buffer memory is managed by the target application, the automatic writing of the buffer to the log files on the disk will not occur if there is an access violation or some other non-recoverable error in the target application. In such cases, you should use this command to manually flush the buffer to the disk, or else the most recently-logged APIs may not appear in the log files.

!logexts.logm

Displays or creates a module inclusion/exclusion list. It is often desirable to only log those API calls that are made from a certain module or set of modules. To facilitate that, Logger allows you to specify a module inclusion list or, alternatively, a module exclusion list. For instance, you would use an inclusion list if you only wanted to log calls from one or two modules. If you wanted to log calls made from all modules except a short list of modules, you would use an exclusion list. The modules Logexts.dll and Kernel32.dll are always excluded, since Logger is not permitted to log itself.

Using Logger.exe

12/5/2018 • 3 minutes to read • Edit Online

One way to activate Logger is to run the stand-alone Logger.exe program. This is essentially a very small debugger that can only take a single target. To run it, include the name of the target application on the command line:

```
logger Target
```

When this is activated, it will load the specified application, and insert code into the target application that will jump off to a routine that loads and initializes Logexts.dll in the target application process. This is referred to as "injecting Logger into the target application."

The Logger.exe utility and the Logexts.dll module are the two components of this Logger vehicle. They communicate through a shared section of memory that includes the output file handles, current category mask, and a pointer to the log output buffer.

A window entitled **Logger (debugger)** will appear. This window will display the progress of Logger.

Change Settings Dialog Box

After the initialization finishes and the initial display is complete, the **Change Settings** dialog box will appear. This allows you to configure the Logger settings. The various settings are described here:

API Settings

This list displays the available API categories. The highlighted categories will be logged; the non-highlighted categories will not. The first time you run Logger, all categories will be highlighted. However, on subsequent runs, Logger will keep track of which categories are selected for a given target application.

If a category is disabled, the hooks for all APIs in that category will be removed so that there is no longer any performance overhead. COM hooks are not removed because they cannot be re-enabled at will.

Enabling only certain categories can be useful when you are only interested in a particular type of interaction that the program is having with Windows -- for example, file operations. This reduces the log file size and also reduces the effect that Logger has on the execution speed of the process.

Logging

This section contains **Enable** and **Disable** radio buttons. Disabling logging will cause all API hooks to be removed in an effort to allow the program to run freely. COM hooks are not removed because they cannot be re-enabled at will.

Inclusion / Exclusion List

This section controls the module inclusion/exclusion list. It is often desirable to log only those function calls that are made from a certain module or set of modules. To facilitate that, Logger allows you to specify a module inclusion list or, alternatively, a module exclusion list. For instance, you would use an inclusion list if you only wanted to log calls from one or two module. If you wanted to log calls made from all modules except a short list of modules, you would use an exclusion list. The modules Logexts.dll and Kernel32.dll are always excluded, since Logger is not permitted to log itself.

Flush the Buffer

This button will flush the current output buffer. As a performance consideration, log output is flushed to disk only when the output buffer is full. By default, the buffer is 2144 bytes.

Since the buffer memory is managed by the target application, the automatic writing of the buffer to the log files

on the disk will not occur if there is an access violation or some other non-recoverable error in the target application. In such cases, you should try to activate the target application's window and hit F12 to get this dialog box back, and then press **Flush the Buffer**. If this is not done, the most recently-logged functions might not appear in the log files.

Go
This causes the target application to begin executing.

Running the Target Application

Once you have chosen the settings, click **Go**. The dialog box will close and the target application will begin to run.

If you make the target application's window active and press F12, it will break into Logger. This will cause the target application to freeze and the **Change Settings** dialog box to reappear. You can alter the settings if you wish, and then press **Go** to continue execution.

You can let the target application run for as long as you wish. If it terminates normally or due to an error, the logging will stop and cannot be restarted.

When you wish to exit, select **File | Exit** and click **Yes**. If the target application is still running, it will be terminated.

Limitations of Logger.exe

When you are running Logger through the Logger.exe tool, it will create only one output file -- an .lgv file. No text file will be written. However, a .txt file of size zero will be created; this could overwrite a text log written by the debugger previously.

The output file will always be placed in LogExts subdirectory of the desktop; this location cannot be changed.

These limitations will not apply if you are running Logger through the debugger and Logexts.dll.

Logger Restrictions and Limitations

12/5/2018 • 2 minutes to read • Edit Online

Logger increases stack consumption for a process because it introduces an additional "wrapping" function before the actual function call.

This can expose bugs in applications that are usually related to uninitialized variables. Since Logger alters stack usage, a local variable declared in a function call might take a different initial value than it does without the presence of Logger. If the program uses this variable without initializing it, the program might crash or otherwise behave differently than if Logger was not present.

Unfortunately, there is no easy way around such problems. The only workaround is to try disabling categories of functions in an attempt to isolate the area that is causing the problem.

LogViewer

12/5/2018 • 2 minutes to read • Edit Online

The LogViewer utility can manipulate an .lgv file, which is a compressed log file produced by the Logger tool. To load a file, simply launch Logviewer.exe through Windows Explorer or from a Command Prompt window. It will take a moment to parse the manifest files. Once this is complete, you can invoke **File | Open** and select the desired .lgv file.

Once LogViewer has opened an .lgv file, it will display a list of all functions in the order they were logged. LogViewer supports powerful commands to filter out functions that you are not interested in. It can also save a filtered version of the log file as a text file.

This section includes:

Reading the LogViewer Display

Filtering the LogViewer Function List

Exporting LogViewer Files to Text

Reading the LogViewer Display

3/6/2019 • 2 minutes to read • Edit Online

LogViewer displays a list of all functions in the order they were logged.

Each row of the display contains several columns. The significance of each column is as follows.

COLUMN	MEANING
+/-	If this column contains a "+" (plus sign), it indicates that the function takes one or more parameters. To see the parameters and their values, either double-click the row or hit the right arrow key when the row is outlined in red. To hide it again, double click it again or hit the left arrow key when the row is outlined in red. There is also a "d#" value in this column. This indicates the "depth" of the function call (in other words, how deep the call is nested in other logged function calls).
#	The sequential row number of the function call. This is useful when you have filters applied and are interested to know how far apart two function calls are.
Thrd	The thread number on which the function call was made. This number is not a thread ID, but is rather an assigned number based on the order that threads were created in the process.
Caller	The instruction address that made the function call. This is derived from the return address for the call. It is actually the return address minus 5 bytes (the typical size of a **call dword ptr** instruction).
Module	The module that contains the calling instruction.
API Function	The name of the function. The name of the module that contains the function is omitted for brevity.
Return Value	The value returned by the function, if it is not a void function.

Double-clicking on a row in the viewer will expand the row to reveal the parameters to the function and their values "going in" to the function. If they are designated as OUT parameters, their value "coming out" is shown on the right.

You can also use the ENTER key or the right and left arrow keys to expand and collapse rows.

Function calls that returned failure status codes are shaded in pink.

Filtering the LogViewer Function List

12/5/2018 • 2 minutes to read • Edit Online

Logger usually captures some function calls that are not needed for your analysis. These can be filtered out by Logger when it creates the log file. However, this process is not reversible, so it is usually preferable to allow all functions to be logged, and then filter the display in LogViewer.

There are several ways to filter out function calls in LogViewer:

- In the main viewing area, selected a function by clicking on it or using the cursor keys. (When a function is selected, LogViewer outlines it in red.) Then, press the DELETE key, or right-click and select **Hide**. This will hide all instances of that function call from view.
- Select **View | APIs Display**. A dialog box will appear that has three areas. On the right is an alphabetical listing of all functions, and on the left are categorical groupings. You can enable or disable the display of any function by checking or clearing the box to the left of its name.
- Select **View | Modules Display**. A dialog box will appear that allows you to select calling modules; only those functions that were called from these modules will be displayed.
- Select **View | First Level Calls Only**. This will display only calls that have "d0" in the left column. It is often desirable to hide function calls that are made by other logged functions. (For example, it is usually not interesting to know that **ShellExecuteEx** makes thirty different registry calls during its course of execution.)

Exporting LogViewer Files to Text

12/5/2018 • 2 minutes to read • Edit Online

A powerful way to manipulate log files is to export them to text. This allows you to find specific parameter values by using the Find facility of any text editor. Although it is possible for a text file to be generated by Logger directly, you will have more flexibility if you use LogViewer to filter the function calls or add aliasing, and then export this information into a text file.

To create a text file from an .lgv file, open the file in LogViewer, and then select **File | Export to Text**. In the dialog box, choose a file name and location.

There are several options at the bottom of the dialog box:

Export Diff Information
This option will alias all pointers, handle values, and other values that change from execution to execution. Instead of outputting the actual value of a pointer (for example, "0x00123FA2"), LogViewer will output "Pointer." Similarly, handles will be aliased as "HeapHandle1" or some other value that will not register as a *diff* when the two files are compared using a differencing tool.

Include Non-Visible Rows
This option disables whatever filters are currently applied to the view while exporting.

Create a Separate File For Each Thread
This option splits up the text file by thread, making it easier to follow a thread of execution. This is useful if you intend to "diff" two instances of execution.

Export Range
This option specifies the range of rows to export.

The Logger Manifest

12/5/2018 • 2 minutes to read • Edit Online

This section includes:

Overview of the Logging Manifest

Manifest File Placement

Manifest File Format

Overview of the Logging Manifest

12/5/2018 • 2 minutes to read • Edit Online

The logging manifest is the group of "header" files that define the functions and COM interfaces that are intercepted and logged. These are not true C++ header files -- they are in a slightly different format that explicitly declares the information needed by Logger.

For example, the manifest format facilitates the following features:

- Designation of OUT parameters. These are parameters that should be logged both on their way into an function and also on their way out.
- Definition of flag masks. This feature allows LogViewer to break a DWORD flag into its constituent bit labels for easier reading.
- Definition of failure cases. This feature allows LogViewer to shade the rows of functions that have returned a failure status code or another error code. Also, if the function sets the "LastError" value for the thread, LogViewer can store away the error code and expand it to its corresponding human-readable error message.
- Designation of parameters that can be aliased for log differencing. This feature gives LogViewer the option of assigning a constant string to a value that changes from execution to execution like pointers and handles when it exports the data to a file. You can then use a differencing tool to compare two execution logs for discrepancies. If pointers and handle values were not aliased, they would produce uninteresting discrepancies when the two files are compared.

Manifest File Placement

12/5/2018 • 2 minutes to read • Edit Online

The primary manifest file must be named Main.h.

When Logger is running, Main.h must be located in the Manifest subdirectory of the directory containing Logexts.dll.

LogViewer is more flexible than Logger. It will search for Main.h in the following directories in this order:

1. The Manifest directory subordinate to the directory containing Logviewer.exe
2. The WinExt\Manifest directory subordinate to the directory containing logviewer.exe
3. The %WinDir%\System32\Manifest directory
4. The %WinDir%\System\Manifest directory

All additional manifest files must reside in the same directory as Main.h.

Manifest File Format

3/6/2019 • 7 minutes to read • Edit Online

The file format for the manifest files borrows as much from C++ and IDL as possible. As a result, it is fairly easy to take a normal C++ SDK header file and modify it to be a manifest file. The parser fully supports C and C++ style comments to help you organize and document the file.

If you are attempting to add a manifest file or make alterations to an existing file, the best way to do it is to just experiment. When you issue a **!logexts.logi** or **!logexts.loge** command in the debugger, Logger will attempt to parse the manifest files. If it encounters a problem, it will produce an error message which might indicate the mistake.

A manifest file is made up of the following basic elements: module labels, category labels, function declarations, COM interface definitions, and type definitions. Other types of elements exist as well, but these are the most important.

Module Labels

A module label simply declares what DLL exports the functions that are declared thereafter. For example, if your manifest file is for logging a group of functions from Comctl32.dll, you would include the following module label before declaring any function prototypes:

```
module COMCTL32.DLL:
```

A module label must appear before any function declarations in a manifest file. A manifest file can contain any number of module labels.

Category Labels

Similar to a module label, a category label identifies which "category" all subsequent functions and/or COM interfaces belong to. For example, if you are creating a Comctl32.dll manifest file, you can use the following as your category label:

```
category CommonControls:
```

A manifest file can contain any number of category labels.

Function Declarations

A function declaration is what actually prompts Logger to log something. It is nearly identical to a function prototype found in a C/C++ header file. There are a few notable additions to the format, which can be best illustrated by the following example:

```
HANDLE [gle] FindFirstFileA(
       LPCSTR lpFileName,
       [out] LPWIN32_FIND_DATAA lpFindFileData);
```

The function **FindFirstFileA** takes two parameters. The first is *lpFileName*, which is a full path (usually with wildcards) defining where to search for a file or files. The second is a pointer to a WIN32_FIND_DATAA structure that will be used to contain the search results. The returned HANDLE is used for future calls to **FindNextFileA**. If **FindFirstFileA** returns INVALID_HANDLE_VALUE, then the function call failed and an error code can be procured by calling the **GetLastError** function.

The HANDLE type is declared as follows:

```
value DWORD HANDLE
{
#define NULL                        0 [fail]
#define INVALID_HANDLE_VALUE      -1 [fail]
};
```

If the value returned by this function is 0 or -1 (0xFFFFFFFF), Logger will assume that the function failed, because such values have a [fail] modifier in the value declaration. (See the Value Types section later in this section.) Since there is a [gle] modifier right before the function name, Logger recognizes that this function uses **GetLastError** to return error codes, so it captures the error code and logs it to the log file.

The [out] modifier on the *lpFindFileData* parameter informs Logger that the data structure is filled in by the function and should be logged when the function returns.

COM Interface Definitions

A COM interface is basically a vector of functions that can be called by a COM object's client. The manifest format borrows heavily from the Interface Definition Language (IDL) used in COM to define interfaces.

Consider the following example:

```
interface IDispatch : IUnknown
{
    HRESULT GetTypeInfoCount( UINT pctinfo  );

    HRESULT GetTypeInfo(
        UINT iTInfo,
        LCID lcid,
        LPVOID ppTInfo );

    HRESULT GetIDsOfNames(
        REFIID riid,
        LPOLECHAR* rgszNames,
        UINT cNames,
        LCID lcid,
        [out] DISPID* rgDispId );

    HRESULT Invoke(
        DISPID  dispIdMember,
        REFIID  riid,
        LCID  lcid,
        WORD  wFlags,
        DISPPARAMS*  pDispParams,
        VARIANT*  pVarResult,
        EXCEPINFO*  pExcepInfo,
        UINT*  puArgErr );
};
```

This declares an interface called **IDispatch** that is derived from **IUnknown**. It contains four member functions, which are declared in specific order within the interface's braces. Logger will intercept and log these member functions by replacing the function pointers in the interface's vtable (the actual binary vector of function pointers used at run time) with its own. See the COM_INTERFACE_PTR Types section later in this section for more details on how Logger captures interfaces as they are handed out.

Type Definitions

Defining data types is the most important (and most tedious) part of manifest file development. The manifest language allows you to define human-readable labels for numeric values that are passed in or returned from a function.

For example, Winerror.h defines a type called "WinError" that is a list of error values returned by most Microsoft Win32 functions and their corresponding human-readable labels. This allows Logger and LogViewer to replace uninformative error codes with meaningful text.

You can also label individual bits within a bit mask to allow Logger and LogViewer to break a DWORD bit mask into its components.

There are 13 basic types supported by the manifest. They are listed in the following table.

TYPE	LENGTH	DISPLAY EXAMPLE
Pointer	4 bytes	0x001AF320
VOID	0 bytes	
BYTE	1 byte	0x32
WORD	2 bytes	0x0A23
DWORD	4 bytes	-234323
BOOL	1 byte	**TRUE**
LPSTR	Length byte plus any number of characters	"Quick brown fox"
LPWSTR	Length byte plus any number of Unicode characters	"Jumped over the lazy dog"
GUID	16 bytes	{0CF774D0-F077-11D1-B1BC-00C04F86C324}
COM_INTERFACE_PTR	4 bytes	0x0203404A
value	Dependent on base type	ERROR_TOO_MANY_OPEN_FILES
mask	Dependent on base type	WS_MAXIMIZED | WS_ALWAYSONTOP
struct	Dependent on size of encapsulated types	+ lpRect nLeft 34 nRight 54 nTop 100 nBottom 300

Type definitions in manifest files work like C/C++ typedefs. For example, the following statement defines PLONG as a pointer to a LONG:

```
typedef LONG *PLONG;
```

Most basic typedefs have already been declared in Main.h. You should only have to add typedefs that are specific to your component. Structure definitions have the same format as C/C++ struct types.

There are four special types: value, mask, GUID, and COM_INTERFACE_PTR.

Value Types
A value is a basic type that is broken out into human-readable labels. Most function documentation only refers to the **#define** value of a particular constant used in a function. For example, most programmers are unaware of what the actual value is for all the codes returned by **GetLastError**, making it unhelpful to see a cryptic numerical value in LogViewer. The manifest value overcomes this by allowing value declarations like as in the following example:

```
value LONG ChangeNotifyFlags
{
#define SHCNF_IDLIST      0x0000        // LPITEMIDLIST
#define SHCNF_PATHA       0x0001        // path name
#define SHCNF_PRINTERA    0x0002        // printer friendly name
#define SHCNF_DWORD       0x0003        // DWORD
#define SHCNF_PATHW       0x0005        // path name
#define SHCNF_PRINTERW    0x0006        // printer friendly name
};
```

This declares a new type called "ChangeNotifyFlags" derived from LONG. If this is used as a function parameter, the human-readable aliases will be displayed instead of the raw numbers.

Mask Types
Similar to value types, a mask type is a basic type (usually a DWORD) that is broken out into human-readable labels for each of the bits that have meaning. Take the following example:

```
mask DWORD DirectDrawOptSurfaceDescCapsFlags
{
#define DDOSDCAPS_OPTCOMPRESSED                 0x00000001
#define DDOSDCAPS_OPTREORDERED                  0x00000002
#define DDOSDCAPS_MONOLITHICMIPMAP              0x00000004
};
```

This declares a new type derived from DWORD that, if used as a function parameter, will have the individual values broken out for the user in LogViewer. So, if the value is 0x00000005, LogViewer will display:

```
DDOSDCAPS_OPTCOMPRESSED | DDOSDCAPS_MONOLITHICMIPMAP
```

GUID Types
GUIDs are 16-byte globally-unique identifiers that are used extensively in COM. They are declared in two ways:

```
struct __declspec(uuid("00020400-0000-0000-C000-000000000046")) IDispatch;
```

or

```
class __declspec(uuid("11219420-1768-11D1-95BE-00609797EA4F")) ShellLinkObject;
```

The first method is used to declare an interface identifier (IID). When displayed by LogViewer, "IID_" is appended to the beginning of the display name. The second method is used to declare a class identifier (CLSID). LogViewer

appends "CLSID_" to the beginning of the display name.

If a GUID type is a parameter to a function, LogViewer will compare the value against all declared IIDs and CLSIDs. If a match is found, it will display the IID friendly name. If not, it will display the 32-hexadecimal-character value in standard GUID notation.

COM_INTERFACE_PTR Types
The COM_INTERFACE_PTR type is the base type of a COM interface pointer. When you declare a COM interface, you are actually defining a new type that is derived from COM_INTERFACE_PTR. As such, a pointer to such a type can be a parameter to a function. If a COM_INTERFACE_PTR basic type is declared as an OUT parameter to a function and there is a separate parameter that has an [iid] label, Logger will compare the passed in IID against all declared GUIDs. If there is a match and a COM interface was declared that has the same name as the IID, Logger will hook all the functions in that interface and log them.

Here is an example:

```
STDAPI CoCreateInstance(
  REFCLSID rclsid,     //Class identifier (CLSID) of the object
  LPUNKNOWN pUnkOuter, //Pointer to controlling IUnknown
  CLSCTX dwClsContext, //Context for running executable code
  [iid] REFIID riid,   //Reference to the identifier of the interface
  [out] COM_INTERFACE_PTR * ppv
                       //Address of output variable that receives
                       //the interface pointer requested in riid
);
```

In this example, *riid* has an [iid] modifier. This indicates to Logger that the pointer returned in *ppv* is a COM interface pointer for the interface identified by *riid*.

It is also possible to declare a function as follows:

```
DDRESULT DirectDrawCreateClipper( DWORD dwFlags, [out] LPDIRECTDRAWCLIPPER *lplpDDClipper, IUnknown *pUnkOuter
);
```

In this example, LPDIRECTDRAWCLIPPER is defined as a pointer to the **IDirectDrawClipper** interface. Since Logger can identify which interface type is being returned in the *lplpDDClipper* parameter, there is no need for an [iid] modifier on any of the other parameters.

PLMDebug

12/5/2018 • 3 minutes to read • Edit Online

PLMDebug.exe is a tool that enables you to use the Windows debugger to debug Windows app, which run under Process Lifecycle Management (PLM). With PLMDebug, you can take manual control of suspending, resuming, and terminating a Windows app.

Tip With Windows 10, version 1607 or later, you can use the UWP commands, such as .createpackageapp to debug UWP apps. For more information see Debugging a UWP app using WinDbg.

Where to get PLMDebug

PLMDebug.exe is included in Debugging Tools for Windows.

```
plmdebug /query [Package]
plmdebug /enableDebug Package [DebuggerCommandLine]
plmdebug /terminate Package
plmdebug /forceterminate Package
plmdebug /cleanterminate Package
plmdebug /suspend Package
plmdebug /resume Package
plmdebug /disableDebug Package
plmdebug /enumerateBgTasks Package
plmdebug /activateBgTaskTaskId
```

Parameters

Package
The full name of a package or the ID of a running process.

DebuggerCommandLine
A command line to open a debugger. The command line must include the full path to the debugger. If the path has blank spaces, it must be enclosed in quotes. The command line can also include arguments. Here are some examples:

```
"C:\Program Files (x86)\Windows Kits\8.0\Debuggers\x64\WinDbg.exe"
```

```
"\"C:\Program Files\Debugging Tools for Windows (x64)\WinDbg.exe\" -server npipe:pipe=test"
```

/query [*Package*]
Displays the running state for an installed package. If *Package* is not specified, this command displays the running states for all installed packages.

/enableDebug *Package* [*DebuggerCommandLine*]
Increments the debug reference count for a package. The package is exempt from PLM policy if it has a non-zero debug reference count. Each call to **/enableDebug** must be paired with a call to /**disableDebug**. If you specify *DebuggerCommandLine*, the debugger will attach when any app from the package is launched.

/terminate *Package*
Terminates a package.

/forceTerminate *Package*
Forces termination of a package.

/cleanTerminate *Package*

Suspends and then terminates a package.

/suspend *Package*
Suspends a package.

/resume *Package*
Resumes a package.

/disableDebug *Package*
Decrements the debug reference count for a package.

/enumerateBgTasks*Package*
Enumerate background task ids for a package.

/activateBgTask*TaskId*
Activates a background task. Note that not all background tasks can be activated using PLMDebug.

Remarks

You must call **plmdebug /enableDebug** before you call any of the suspend, resume, or terminate functions.

The PLMDebug tool calls the methods of the IPackageDebugSettings interface. This interface enables you to take manual control of the process lifecycle management for your apps. Through this interface (and as a result, through this tool), you can suspend, resume, and terminate your Windows app. Note that the methods of the IPackageDebugSettings interface apply to an entire package. Suspend, resume, and terminate affect all currently running apps in the package.

Examples

Example 1

Attach a debugger when your app is launched

Suppose you have an app named MyApp that is in a package named MyApp_1.0.0.0_x64__tnq5r49etfg3c. Verify that your package is installed by displaying the full names and running states all installed packages. In a Command Prompt window, enter the following command.

plmdebug /query

```
Package full name: 1daa103b-74e1-426d-8193-b6bc7ed66fed_1.0.0.0_x86__tnq5r49etfg3c
Package state: Terminated

Package full name: 41fb5f27-7b60-4f5e-8459-803673131dd9_1.0.0.0_x86__tnq5r49etfg3c
Package state: Suspended
...
Package full name: MyApp_1.0.0.0_x64__tnq5r49etfg3c
Package state: Terminated
...
```

Increment the debug reference count for your package, and specify that you want WinDbg to attach when your app is launched.

plmdebug /enableDebug MyApp_1.0.0.0_x64__tnq5r49etfg3c "C:\Program Files (x86)\Windows Kits\8.0\Debuggers\x64\WinDbg.exe"

When you launch your app, WinDbg will attach and break in.

When you have finished debugging, detach the debugger. Then decrement the debug reference count for your package.

plmdebug /disableDebug MyApp_1.0.0.0_x64__tnq5r49etfg3c

Example 2

Attach a debugger to an app that is already running

Suppose you want to attach WinDbg to MyApp, which is already running. In WinDbg, on the **File** menu, choose **Attach to a Process**. Note the process ID for MyApp. Let's say the process ID is 4816.

Increment the debug reference count for the package that contains MyApp.

plmdebug /enableDebug 4816

In WinDbg, in the **Attach to Process** dialog box, select process 4816, and click **OK**. WinDbg will attach to MyApp.

When you have finished debugging MyApp, detach the debugger. Then decrement the debug reference count for the package.

plmdebug /disableDebug 4816

Example 3

Manually suspend and resume your app

Suppose you want to manually suspend and resume your app. First, increment the debug reference count for the package that contains your app.

plmdebug /enableDebug MyApp_1.0.0.0_x64__tnq5r49etfg3c

Suspend the package. Your app's suspend handler is called, which can be helpful for debugging.

plmdebug /suspend MyApp_1.0.0.0_x64__tnq5r49etfg3c

When you have finished debugging, resume the package.

plmdebug /resume MyApp_1.0.0.0_x64__tnq5r49etfg3c

Finally, decrement the debug reference count for the package.

plmdebug /disableDebug MyApp_1.0.0.0_x64__tnq5r49etfg3c

See also

How to trigger suspend, resume, and background events in Windows Apps

Tools Included in Debugging Tools for Windows

Remote Tool

12/5/2018 • 2 minutes to read • Edit Online

The Remote tool, Remote.exe, is a command-line tool that lets you run and control any console program from a remote computer.

Where to get the Remote tool

Remote.exe is included in Debugging Tools for Windows.

Remote tool components

The Remote tool includes the following components:

- A server application that starts a console program and opens a named pipe for client connections.
- A client application that establishes a connection to a server. Commands typed on the client computer are sent to the console application on the server, and the remote client displays the output from the server's console window.
- A query feature that lists the remote sessions running on a server computer.

With the Remote tool, you can start multiple server sessions on a single computer where multiple clients can connect to each session. The sessions are limited only by the computer's resources.

This is an older tool that has been eclipsed by more sophisticated tools, primarily Remote Desktop. However, because it is simple and uses very few resources, it is still widely used, typically for remote debugging.

The Remote tool requires that commands be submitted on both the local and remote computers. As such, to use this tool on a computer without a local user, you must develop an alternate way to submit the commands and to restart the computer, if necessary.

The Remote tool can compromise security on your computer, because it does not authenticate users or use Microsoft Windows permissions. By default, any remote computer that can connect and provide the session name can use the named pipe that this tool creates, although you can use Remote tool options to include and exclude particular users and groups.

This section includes:

Remote Tool Concepts

Remote Tool Commands

Remote Tool Examples

Related topics

Tools Included in Debugging Tools for Windows

Remote Tool Concepts

12/5/2018 • 2 minutes to read • Edit Online

The following concepts are used in the Remote tool.

Client and Server

The Remote tool uses a client-server paradigm that avoids the words *local* and *remote*, which are relative terms that can be confusing when there are multiple users and multiple computers.

Commands that you type at the client and server computers appear in the Command Prompt windows of both computers.

The Server

The *server* is the computer on which the console program runs. The *Remote Server* is the instance of the Remote tool running on the server. The server establishes and names the remote session (named pipe), issues the command to start the console program, and determines who is permitted to connect to the session.

The Client

The *client* is a remote computer that sends commands to the console program. The *Remote Client* is the instance of the Remote tool running on the client computer. The client connects to the remote session that the server established and uses the remote session (named pipe) that the server created to send commands to the console program that runs on the server.

The Remote tool supports multiple clients in each remote session. Each client is running one Remote Client. All of the clients can send commands to the console program running on the server, and all of the clients can see the commands sent and output displayed.

Visible Session

When remote sessions are *visible*, they appear in the list of remote sessions on the computer. To display the list, use the **Remote Server query command** (**/q**).

By default, only debugger sessions are visible, that is, sessions in which the value of the *Command* parameter include the words **kd**, **dbg**, **remoteds**, **ntsd**, or **cdb**; otherwise, the session is not visible. The *Command* parameter is part of the Remote tool command on the server.

To make a session visible, add the **/v** parameter to the **Remote Server** command. To make a debugger session invisible, add the **/-v** parameter to the command.

For help with the Remote Server query command, see **Remote Server Query Command**.

Remote Tool Commands

12/5/2018 • 2 minutes to read • Edit Online

The Remote tool has a separate command syntax for the client and server sides of a session. Commands to start a server session must be entered first because they establish the communications pipe to which the client connects.

The Remote tool also has a query command and a separate set of commands that you use to communicate with the Remote tool (instead of the console program) during a remote session.

These commands are described in the following topics:

Remote Server Syntax

Remote Client Syntax

Remote Server Query Command

Remote Session Commands

Remote Server Syntax

12/5/2018 • 2 minutes to read • Edit Online

To start the server side of the Remote tool, use the following syntax at the command line.

```
remote /s Command SessionName [/f Color] [/b] [/u User [/u User...]] [/ud User [/ud User...]] [/v | /-v]
```

Parameters

/s
Starts a server session.

Command
Specifies the command that starts the console-based program. The command can include parameters. If the command includes spaces, enclose it in quotation marks.

SessionName
Assigns a name to the remote session. If the name includes spaces, enclose it in quotation marks. This parameter is not case-sensitive.

/f
Specifies the color of the text in the server command window.

/b
Specifies the background color of the server command window.

Color
Specifies a color. Valid values are black, blue, green, cyan, red, purple, yellow, white, lblack, lblue, lgreen, lred, lpurple, lyellow, and lwhite.

/u
Specifies users or groups that are permitted to connect to the remote session; by default, everyone is permitted. When you use this parameter, everyone is denied permission, except for the users and groups specified by the *User* subparameter.

/ud
Specifies users or groups that are denied permission to connect to the remote session; by default, no one is denied permission.

User
Specifies the name of a user or group in [*domain* | *computer*]\{*user* | *group*} format. When specifying users or groups on the local computer, omit the computer name.

/v
Makes a session visible. For more information, see Visible Session.

By default, only debugger sessions are visible, that is, sessions in which the value of the *Command* parameter include the words **kd**, **dbg**, **remoteds**, **ntsd**, or **cdb**; otherwise, the session is not visible.

-v
Makes a remote debugger session invisible. For more information, see Visible Session.

Comments

The *Command* and *SessionName* parameters must appear in the order shown on the syntax line.

To end a remote session, type **@k**. For more information, see Remote Session Commands.

The Remote tool will not create a session that the current user does not have permission to join.

When starting more than one remote session on a single computer, open a new command window for each session. Also, use a different session name for each session. Because the session names are used to label the named pipes, they must be unique on the computer.

Sample Usage

```
remote /s "i386kd -v" TestSession
remote /s "cmd" "My Remote Session" /f white /b black /u Server01\Administrators
remote /s "ntsd -d -g -x" DebugIt /-v
remote /q Server01
```

Remote Client Syntax

10/24/2019 • 2 minutes to read • Edit Online

To start the client side of the Remote tool, use the following syntax at the command line.

```
remote /c Server SessionName [/L Lines] [/f] [/b] [/k ColorFile]
```

Parameters

/c
Connects the client to a remote session.

Server
Specifies the computer name of the server that established the session.

SessionName
Specifies the name of the remote session. This parameter is not case-sensitive.

/L *Lines*
Specifies the number of lines from the console display that are sent to the client computer. The default is 200 lines. *Lines* is a decimal number.

/f
Specifies the color of the text in the server command window.

/b
Specifies the background color of the server command window.

Color
Specifies a color. Valid values are black, blue, green, cyan, red, purple, yellow, white, lblack, lblue, lgreen, lred, lpurple, lyellow, and lwhite.

/k *ColorFile*
Indicates the path (optional) and name of a formatted text file that specifies the colors for displaying the output on the client computer.

The color file associates keywords in the output with text colors. When the keywords appear in a line of output, the Remote tool applies the associated text color to that line. For the format of the file, see "Remarks".

/q *Computer*
Displays the remote sessions available on the specified computer. Only visible sessions appear in the list. (See **/v** in **Remote Server Syntax**.)

Comments

The *Server* and *SessionName* parameters must appear in the order shown on the syntax line.

To disconnect from a remote session, type **@q**. For more information, see Remote Session Commands.

Keyword Color File. The format of the keyword color file is as follows. The keyword interpreter is not case sensitive.

The keyword or phrase appears on a line by itself. The colors associated with that keyword appear by themselves

on the following line, as shown in the syntax:

```
Keyword
TextColor[, BackgroundColor]
```

For example, the following file directs Remote to display lines that include the word "error" in black text on a light red background; to display lines that include the word "warning" in light blue (on the default background), and lines that include the phrase "Windows Vista" in light green on the default background.

```
ERROR
black, lred
WARNING
lblue
Windows Vista
lgreen
```

Sample Usage

```
remote /c Server01 TestSession
remote /c Domain1\ComputerA0 "cmd" "My Remote Session"
remote /c Server01 TestSession /L 50 /f black /b white /k c:\remote_file.txt
remote /q Server01
```

Remote Server Query Command

12/5/2018 • 2 minutes to read • Edit Online

To display a list of available sessions on a local or Remote Server, use the following syntax.

```
remote /q Computer
```dbgcmd

## <span id="ddk_remote_server_query_command_dtools"></span><span id="DDK_REMOTE_SERVER_QUERY_COMMAND_DTOOLS">
</span>Parameters


<span id="________q______"></span><span id="________Q______"></span> **/q**
Query. Displays the visible remote sessions on the specified computer.

<span id="_______Computer______"></span><span id="_______computer______"></span><span
id="_______COMPUTER______"></span> *Computer*
Specifies the name of the computer. This parameter is required (even on the local computer).

### <span id="comments"></span><span id="COMMENTS"></span>Comments

The query display includes only server sessions; it does not display client connections to remote sessions.

The query display includes only visible sessions. There is no command to display invisible sessions.

The query display includes all visible sessions, including those that restrict users (**/u** and **/ud**). The
user might not have permission to connect to all of the sessions in the list.

When there are no remote sessions running on the server, the Remote tool displays the following message:

```console
No Remote servers running on \\Computer
```

However, when the only remote sessions running on the computer are invisible remote sessions, the Remote tool displays the following message:

```
No visible sessions on server \\Computer
```

Remote Session Commands

12/5/2018 • 2 minutes to read • Edit Online

Use the following commands to communicate with the Remote tool during a console session.

@H
Displays the session commands on server and client computers. Works on server and client computers.

@M *Message*
Displays the specified message on all server and client computers in the session. Works on server and client computers.

@P *Message*
Generates a pop-up window on the server computer with the specified message. On client computers, the message appears in the command window. Works on server and client computers.

@Q
Quit. Disconnects a client computer from the session. Works only on a client computer.

@K
Disconnects all clients and ends the remote session. Works only on a server computer.

Comments

For samples, see "Using the Session Commands" in Remote Tool Examples.

In response to the session commands, the Remote tool displays a label with the day and time that the command was executed. The time for all events is displayed in the time zone of the server computer.

Remote Tool Examples

12/5/2018 • 6 minutes to read • Edit Online

The examples in this section demonstrate the use of the Remote tool and show sample input and output.

Basic Server Command

The following command starts a remote session on the computer.

The command uses the **/s** parameter to indicate a server-side command. It uses the command, **cmd**, to start the Windows command shell (Cmd.exe), and names the session **test1**.

```
remote /s cmd test1
```

In response, the Remote tool starts the session and displays the command that clients would use to connect to the session.

```
**************************************
***********     REMOTE    ************
***********     SERVER    ************
**************************************
To Connect: Remote /C SERVER06 "test1"

Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.
```

Basic Client Command

The following command connects to a remote session on the Server01 computer. The command uses the **/c** parameter to indicate a client-side command. It specifies the name of the server computer, **Server01**, and the name of the session on that computer, **test1**.

```
remote /c server01 test1
```

In response, the Remote tool displays a message reporting that the client computer is connected to the session on the server computer. The message displays the name of the server computer and local user (**Server04 user1**).

```
**************************************
***********     REMOTE    ************
***********     CLIENT    ************
**************************************
Connected...

Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Program Files\Debugging Tools for Windows>
**Remote: Connected to SERVER04 user1 [Tue 9:39 AM]
```

After the client connects to the server, the commands typed at the command prompt on the client and server computers appear on both displays..

For example, if you type **dir** at the command prompt of the client computer, the directory display appears in the Command Prompt window on both the client and server computers.

Using Server Options

The following server-side command starts a remote session with the NTSD debugger.

The command uses the **/s** parameter to indicate a server-side command. The next parameter, **"ntsd -d -v"**, is the console command that starts the debugger, along with the debugger options. Because the console command includes spaces, it is enclosed in quotation marks. The command includes a name for the session, **debugit**.

The command uses the **/u** parameter to permit only administrators of the computer and a particular user, User03 in Domain01, to connect to the session. It uses the **/f** and **/b** options to specify black text (foreground) on a white background.

Finally, the command uses the **/-v** parameter to make the session invisible to user queries. Debugger sessions are visible by default.

```
remote /s "ntsd -d -v" DebugIt /u Administrators /u Domain01\User03
/f black /b white /-v
```

In response, the Remote tool creates a session named DebugIt and starts NTSD with the specified parameters. The message indicates that only the specified users have permission to connect. It also changes the command window to the specified colors.

```
**************************************
***********     REMOTE    ************
***********     SERVER    ************
**************************************

Protected Server!  Only the following users or groups can connect:
    Administrators
    Domain01\User03
To Connect: Remote /C SERVER06 "debugit"

Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.
```

Using Client Options

The following command connects to the remote session with the NTSD debugger that was started in the previous example.

The command uses the **/c** parameter to indicate a client-side command. It specifies the name of the server computer, **server06**, and the name of the remote session, **debugit**.

The command also includes the **/k** parameter to specify the location of a keyword color file.

```
remote /c server06 debugit /k c:\remote_client.txt
```

The color file includes the following text:

```
Registry
white, blue
Token
red, white
```

This text instructs the Remote tool to display lines of output with the word "registry" (not case sensitive) in white text on a blue background and to display lines of output with the word "token" in red text on a white background.

In response, the Remote tool connects the client to the server session and displays the following message.

```
************************************
***********     REMOTE    ************
***********     CLIENT    ************
************************************
Connected...

Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.
```

The client can now send commands to the NTSD debugger on the server computer. The output from the command appears on both the client and server computers.

Lines of output with the word "registry" are displayed on the client computer in white text on a blue background, and lines of output with the word "kernel" in red text on a white background.

Querying a Session

The Remote tool includes a query parameter (**/q**) that displays a list of remote sessions on a particular computer. The display includes only visible sessions (debugger sessions started without the **/-v** parameter and non-debugger sessions started with the **/v** parameter).

You can query for sessions from the server or client computers. You must specify the computer name, even when querying for sessions on the local computer.

The following command queries for sessions on the local computer, **Server04**.

```
remote /q Server04
```

In response, the Remote tool reports that there are no remote sessions running on the local computer.

```
Querying server \\Server04
No Remote servers running on \\Server04
```

In contrast, in response to a query about sessions on a different computer, **Server06**, the Remote tool lists the sessions running on that computer.

```
Querying server \\Server06

Visible sessions on server Server06:

ntsd                            [Remote /C SERVER06 "debug"] visible
cmd                             [Remote /C SERVER06 "test"] visible
```

The display lists the visible sessions, the console programs running on those sessions (NTSD and the Command Prompt window), and the command that connects to the session. The session name appears in the command syntax in quotation marks.

The display does not show the permissions established for these sessions, if any. Therefore, the display might include sessions that you do not have permission to join.

Using the Session Commands

You can use the remote session commands at any time during a remote session.

The following command sends a message to all computers connected to the session.

```
@M I think I found the problem.
```

As a result, the message appears in the Command Prompt windows of all computers in the session. The message includes the computer name and the day and time of the message.

```
@m I think I found the problem.     [SERVER01       Wed 11:53 AM]
```

When the message is sent from the server computer, "Local" appears in the label instead of the computer name.

```
@m I think I found the problem.     [Local       Wed 11:52 AM]
```

The following command generates a pop-up message that appears on the server computer. On all client computers in the session, it writes a message to the Command Prompt window.

```
@P Did you see that?
```

On client computers, the pop-up message appears in the command window.

```
From SERVER02  [Wed 11:58 AM]

 Did you see that?
```

The time that appears in the message label is always the time on the server computer, even if the client computer that sent the message is in a different time zone.

Ending a Remote Session

The following examples demonstrate how to use the remote session commands to disconnect a client computer from a session and to end a remote session. Only the server computer that started the remote session can end it.

To disconnect a client computer from a remote session, on the client computer, type **@q**.

In response, the following message appears on the client computer that disconnected.

```
*** SESSION OVER ***
```

On all other computers in the session, the Remote tool posts a message with the name of the computer and user who disconnected, and the day and time of the disconnect.

```
**Remote:  Disconnected from SERVER04 User01  [Wed 12:01 PM]
```

To end a remote session, on the server computer, type **@k**. This command automatically disconnects the clients, and then ends the session.

TList

12/5/2018 • 2 minutes to read • Edit Online

TList (Task List Viewer), Tlist.exe, is a command-line tool that displays the processes running on the local computer along with useful information about each process.

TList displays:

- All processes running on the computer, along with their process IDs (PIDs).
- A tree showing which processes created each process.
- Details of the process, including its virtual memory use and the command that started the process.
- Threads running in each process, including their thread IDs, entry points, last reported error, and thread state.
- The modules running in each process, including the version number, attributes, and virtual address of the module.

You can use TList to search for a process by name or PID, or to find all processes that have loaded a specified module.

In Windows XP and later versions of Windows, TList was replaced by TaskList (Tasklist.exe), which is described in Help and Support for those systems. TList is included in Debugging Tools for Windows to support users who do not have access to TaskList.

This section includes:

TList Commands

TList Examples

TList Commands

12/5/2018 • 2 minutes to read • Edit Online

The syntax of the TList command is as follows:

```
tlist [/p ProcessName | PID | Pattern | /t | /c | /e | /k | /m [Module] | /s | /v
```

Parameters

tlist
Without additional parameters, TList displays all running processes, their process identifiers (PIDs), and the title of the window in which they are running, if any.

/p *ProcessName*
Displays the process identifier (PID) of the specified process.

ProcessName is the name of the process (with or without file name extension), not a pattern.

If the value of *ProcessName* does not match any running process, TList displays -1. If it matches more than one process name, TList displays only the PID of the first matching process.

PID
Displays detailed information about the process specified by the PID. For information about the display, see the "Remarks" section below. To find a process ID, type **tlist** without additional parameter.

Pattern
Displays detailed information about all processes whose names or window titles match the specified pattern. Pattern can be a complete name or a regular expression.

/t
Displays a task tree in which each process appears as a child of the process that created it.

/c
Displays the command line that started each process.

/e
Displays the session identifier for each process.

/k
Displays the COM components active in each process.

/m *Module*
Lists tasks in which the specified DLL or executable module is loaded. Module can be a complete module name or a module name pattern.

/s
Displays the services that are active in each process.

/v
Displays details of running processes including the process ID, session ID, window title, command line, and the services running in the process.

Comments

In its detailed display of a process (**tlist** *PID* or **tlist** *Pattern*), TList displays the following information.

- Process ID, executable name, friendly name of the program.
- Current working directory (CWD).
- The command line that started the process (CmdLine).
- Current virtual address space values.
- Number of threads.
- A list of threads running in the process. For each thread, TList displays the thread ID (TID), the function that the thread is running, the address of the entry point, the address of the last reported error (if any), and the thread state.
- A list of the modules loaded for the process. For each module, TList displays the version number, attributes, virtual address of the module, and the module name.

When using the **/e** parameter, valid session identifiers appear in the process list only under the following conditions. Otherwise, the session identifier is zero (0).

- On Windows 2000 and Windows Server 2003, at least one user must be connected to a session other than the console session.
- On Windows XP, Fast User Switching must be enabled and more than one user must be connected to the non-console session.
- On Windows Vista, where all processes are associated with two Terminal Services sessions by default, at least one user must be connected to the non-console session.

TList Examples

12/5/2018 • 3 minutes to read • Edit Online

The following examples demonstrate how to use TList.

Simplest TList Command (tlist)

Typing **tlist** without additional parameters displays a list of running processes, their process IDs (PIDs), and the title of the window in which they are running, if any.

```
c:\>tlist

   0 System Process
   4 System
 308 smss.exe
 356 csrss.exe
 380 winlogon.exe      NetDDE Agent
 424 services.exe
 436 lsass.exe
 604 svchost.exe
 776 svchost.exe
 852 spoolsv.exe
1000 clisvcl.exe
1036 InoRpc.exe
1064 InoRT.exe
1076 InoTask.exe
1244 WTTSvc.exe
1492 Sysparse_com.exe  OleMainThreadWndName
1980 explorer.exe      Program Manager
1764 launch32.exe      SMS Client User Application Launcher
1832 msmsgs.exe        MSBLNetConn
2076 ctfmon.exe
2128 ISATRAY.EXE       IsaTray
4068 tlist.exe
```

Find a process ID (-p)

The following command uses the **-p** parameter and process name to find the process ID of the Explorer.exe (Explorer) process.

In response, TList displays the process ID of the Explorer process, 328.

```
c:\>tlist -p explorer
328
```

Find process details using PID

The following command uses the process ID of the process in which Explorer is running to find detailed information about the Explorer process.

```
c:\>tlist 328
```

In response, TList displays details of the Explorer process including the following elements:

- Process ID, executable name, program friendly name.
- Current working directory (CWD).

- The command line that started the process (CmdLine).
- Current virtual address space values.
- Number of threads.
- A list of threads running in the process. For each thread, TList displays the thread ID (TID), the function that the thread is running, the address of the entry point, the address of the last reported error (if any), and the thread state.
- A list of the modules loaded for the process. For each module, TList displays the version number, attributes, virtual address of the module, and the module name.

The following is an excerpt of the output resulting from this command.

```
 328 explorer.exe      Program Manager
   CWD:     C:\Documents and Settings\user01\
   CmdLine: C:\WINDOWS\Explorer.EXE
   VirtualSize:    90120 KB   PeakVirtualSize:   104844 KB
   WorkingSetSize: 19676 KB   PeakWorkingSetSize: 35716 KB
   NumberOfThreads: 17
    332 Win32StartAddr:0x010160cc LastErr:0x00000008 State:Waiting
   1232 Win32StartAddr:0x70a7def2 LastErr:0x00000000 State:Waiting
   1400 Win32StartAddr:0x77f883de LastErr:0x00000000 State:Waiting
   1452 Win32StartAddr:0x77f91e38 LastErr:0x00000000 State:Waiting
   1484 Win32StartAddr:0x70a7def2 LastErr:0x00000006 State:Waiting
   1904 Win32StartAddr:0x74b02ed6 LastErr:0x00000000 State:Ready
   1948 Win32StartAddr:0x72d22ecc LastErr:0x00000000 State:Waiting
   ....  (thread data deleted here)

  6.0.2800.1106 shp  0x01000000  Explorer.EXE
  5.1.2600.1217 shp  0x77F50000  ntdll.dll
  5.1.2600.1106 shp  0x77E60000  kernel32.dll
  7.0.2600.1106 shp  0x77C10000  msvcrt.dll
  5.1.2600.1106 shp  0x77DD0000  ADVAPI32.dll
  5.1.2600.1254 shp  0x78000000  RPCRT4.dll
  5.1.2600.1106 shp  0x77C70000  GDI32.dll
  5.1.2600.1255 shp  0x77D40000  USER32.dll
  ....  (module data deleted here)
```

Find multiple processes (Pattern)

The following command searches for processes by a regular expression that represents the process name or window name of one or more processes. In this example, the command searches for a process whose process name or window name begins with "ino."

```
c:\>tlist ino*
```

In response, TList displays process details for Inorpc.exe, Inort.exe, and Inotask.exe. For a description of the output, see the "Find process details using PID" subsection above.

Display a process tree (/t)

The following command displays a tree that represents the processes running on the computer. Processes appear as the children of the process that created them.

```
c:\>tlist /t
```

The resulting process tree follows. This tree shows, among other things, that the System (4) process created the Smss.exe process, which created Csrss.exe, Winlogon.exe, Lsass.exe and Rundll32.exe. Also, Winlogon.exe created

Services.exe, which created all of the service-related processes.

```
System Process (0)
System (4)
  smss.exe (404)
    csrss.exe (452)
    winlogon.exe (476) NetDDE Agent
      services.exe (520)
        svchost.exe (700)
        svchost.exe (724)
        svchost.exe (864)
        svchost.exe (888)
        spoolsv.exe (996)
        scardsvr.exe (1040)
        alg.exe (1172)
        atievxx.exe (1200) ATI video bios poller
        InoRpc.exe (1248)
        InoRT.exe (1264)
        InoTask.exe (1308)
        mdm.exe (1392)
        dllhost.exe (2780)
      lsass.exe (532)
      rundll32.exe (500)
explorer.exe (328) Program Manager
  WLANMON.exe (1728) TI Wireless LAN Monitor
  ISATRAY.EXE (1712) IsaTray
  cmmon32.exe (456)
  WINWORD.EXE (844) Tlist.doc - Microsoft Word
  dexplore.exe (2096) Platform SDK - CreateThread
```

Find process by module (/m)

The following command finds all of the processes running on the computer that load a particular DLL.

```
c:\>tlist /m
```

In response, TList displays process details for Inorpc.exe, Inort.exe, and Inotask.exe. For a description of the output, see the "Find process details using PID" subsection above.

UMDH

12/5/2018 • 2 minutes to read • Edit Online

The User-Mode Dump Heap (UMDH) tool, Umdh.exe, analyzes the Microsoft Windows heap memory allocations for a given process. UMDH has the following modes.

- **Analyze a running process** ("Mode 1"). UMDH captures and analyzes the heap memory allocations for a process. For each allocation, UMDH displays the size of the allocation, the size of the overhead, the pointer to the allocation and the allocation stack. If a process has more than one active memory heap, UMDH captures all heaps. This analysis can be displayed in realtime or saved in a log file.
- **Analyze UMDH log files** ("Mode 2"). UMDH analyzes the log files it has previously created. UMDH can compare two logs created for the same process at different times, and display the calls in which the allocation size increased the most. This technique can be used to find memory leaks.

This section includes:

Preparing to Use UMDH

UMDH Commands

Interpreting a UMDH Log

Interpreting a Log Comparison

Preparing to Use UMDH

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

You must complete the configuration tasks described in this section before using User-Mode Dump Heap (UMDH) to capture the heap allocations for a process. If the computer is not configured correctly, UMDH will not generate any results or the results will be incomplete or incorrect.

Create the user-mode stack trace database

Before using UMDH to capture the heap allocations for a process, you must configure Windows to capture stack traces.

To enable stack trace capturing for a process, use GFlags to set the **Create user mode stack trace database** flag for the process. This can be done by either of the following methods:

- In the GFlags graphical interface, choose the **Image File** tab. Type the process name, including the file name extension (for example, Notepad.exe). Press the **TAB** key, select **Create user mode stack trace database**, and then click **Apply**.
- Or, equivalently, use the following GFlags command line, where *ImageName* is the process name (including the file name extension):

  gflags /i *ImageName* **+ust**

By default, the amount of stack trace data that Windows gathers is limited to 32 MB on an x86 processor, and 64 MB on an x64 processor. If you must increase the size of this database, choose the **Image File** tab in the GFlags graphical interface, type the process name, press the **TAB** key, check the **Stack Backtrace (Megs)** check box, type a value (in MB) in the associated text box, and then click **Apply**.

Note Increase this database only when necessary, because it may deplete limited Windows resources. When you no longer need the larger size, return this setting to its original value.

These settings affects all new instances of the program. It does not affect currently running instances of the program.

Access the Necessary Symbols

Before using UMDH, you must have access to the proper symbols for your application. UMDH uses the symbol path specified by the environment variable _NT_SYMBOL_PATH. Set this variable equal to a path containing the symbols for your application.

If you also include a path to Windows symbols, the analysis may be more complete. The syntax for this symbol path is the same as that used by the debugger; for details, see Symbol Path.

For example, if the symbols for your application are located at C:\MyApp\Symbols, and you have installed the Windows symbol files to \\myshare\winsymbols, you would use the following command to set your symbol path:

```
set _NT_SYMBOL_PATH=c:\myapp\symbols;\\myshare\winsymbols
```

As another example, if the symbols for your application are located at C:\MyApp\Symbols, and you want to use the public Microsoft symbol store for your Windows symbols, using C:\MyCache as your downstream store, you would use the following command to set your symbol path:

```
set _NT_SYMBOL_PATH=c:\myapp\symbols;srv*c:\mycache*https://msdl.microsoft.com/download/symbols
```

Important Suppose you have two computers: a *logging computer* where you create a UMDH log and an *analysis computer* where you analyze the UMDH log. The symbol path on your analysis computer must point to the symbols for the version of Windows that was loaded on the logging computer at the time the log was made. Do not point the symbol path on the analysis computer to a symbol server. If you do, UMDH will retrieve symbols for the version of Windows that is running on the analysis computer, and UMDH will not display meaningful results.

Disable BSTR Caching

Automation (formerly known as OLE Automation) caches the memory used by BSTR strings. This can prevent UMDH from correctly determining the owner of a memory allocation. To avoid this problem, you must disable BSTR caching.

To disable BSTR caching, set the OANOCACHE environment variable equal to one (1). This setting must be made before launching the application whose allocations are to be traced.

Alternatively, you can disable BSTR caching from within the application itself by calling the .NET Framework **SetNoOaCache** function. If you choose this method, you should call this function early, because any BSTR allocations that have already been cached when **SetNoOaCache** is called will remain cached.

If you need to trace the allocations made by a service, you must set OANOCACHE as a system environment variable and then restart Windows for this setting to take effect.

On Windows 2000, in addition to setting OANOCACHE equal to 1, you must also install the hotfix available with Microsoft Support Article 139071. This hotfix is not needed on Windows XP and later versions of Windows.

Find the Process ID

UMDH identifies the process by its process identifier (PID). You can find the PID of any running process by using Task Manager, Tasklist, or TList.

UMDH Commands

12/5/2018 • 2 minutes to read • Edit Online

UMDH has two different modes, each with its own command syntax and parameters:

- **Analyze a Running Process** - Displays all the heap allocation in a process.
- **Analyze UMDH Logs** - Compares the allocation list made at two different times for the same process. Analyzing the differences can help to identify memory leaks.

Analyze a Running Process

1/3/2019 • 2 minutes to read • Edit Online

Use the following commands to record and analyze the heap memory allocations in a running process. This analysis focuses on stack traces.

```
umdh -p:PID [-f:LogFile] [-v[:MsgFile]] | [-g] | [-h]
```

Parameters

-p:*PID*
Specifies the process to analyze. *PID* is the process ID of the process. This parameter is required.

To find the PID of a running process, use Task Manager, Tasklist, or TList.

-f:*LogFile*
Saves the log contents in a text file. By default, UMDH writes the log to stdout (command window).

LogFile specifies the path (optional) and name of the file. If you specify an existing file, UMDH overwrites the file.

Note If UMDH was not started in Administrator mode, or attempts to write to "protected" paths, it will be denied access.

-v[:*MsgFile***]**
Verbose mode. Generates detailed informational and error messages. By default, UMDH writes these messages to stderr.

MsgFile specifies the path (optional) and name of a text file. When you use this variable, UMDH writes the verbose messages to the specified file, instead of to stderr. If you specify an existing file, UMDH overwrites the file.

-g
Logs the heap blocks that are not referenced by the process ("garbage collection").

-h
Displays help.

Comments

On Windows 2000, if UMDH is reporting errors finding the stack trace database, and you have enabled the **Create user mode stack trace database** option in GFlags, you might have a symbol file conflict. To resolve it, copy the DBG and PDB symbol files for the application to the same directory, and try again.

Sample Usage

```
umdh -?
umdh -p:2230
umdh -p:2230  -f:dump_allocations.txt
umdh -p:2230 -f:c:\Log1.txt -v:c:\Msg1.txt
umdh -p:2230 -g -f:garbage.txt
```

Analyze UMDH Logs

12/5/2018 • 2 minutes to read • Edit Online

Use the following commands to analyze User-Mode Dump Heap (UMDH) logs that were created by running UMDH with the syntax described in **Analyze a Running Process**. This analysis focuses on allocations, instead of stack traces.

You can analyze a single log file or compare logs from different runs to detect the changes in the program or driver's memory dump allocations over time.

```
umdh [-d] [-v] [-l] File1 [File2] [-h | ?]
```

Parameters

-d
Displays numeric data in decimal numbers. The default is hexadecimal.

-v
Verbose mode. Includes the traces, as well as summary information. The traces are most helpful when analyzing a single log file.

-l
Includes file names and line numbers in the log. (Please note that the parameter is the lowercased letter "L," not the number one.)

File1 [*File2*]
Specifies the UMDH log files to analyze.

UMDH creates log files when you run it in the **analyze a running process** mode and save the log content in a text file (**-f**).

When you specify one log file, UMDH analyzes the file and displays the function calls in each trace in descending order of bytes allocated.

When you specify two log files, UMDH compares the files and displays in descending order the function calls whose allocations have grown the most between the two trials.

-h | ?
Displays help.

Sample Usage

```
umdh dump.txt
umdh -d -v dump.txt
umdh dump1.txt dump2.txt
```

Remarks

Suppose you have two computers: a *logging computer* where you create a UMDH log and an *analysis computer* where you analyze the UMDH log. The symbol path on your analysis computer must point to the symbols for the version of Windows that was loaded on the logging computer at the time the log was made. Do not point the symbol path on the analysis computer to a symbol server. If you do, UMDH will retrieve symbols for the version of

Windows that is running on the analysis computer, and UMDH will not display meaningful results.

Interpreting a UMDH Log

12/5/2018 • 2 minutes to read • Edit Online

User-Mode Dump Heap (UMDH) log files display the list of heap allocations in the process and the stacks where the allocations were made.

This example shows how to generate a log for a process that has ID 1204. The log is written to the file log1.txt.

```
umdh -p:1204 -f:log1.txt
```

The log file is not readable because the symbols are not resolved. UMDH resolves symbols when you analyze the log. This example shows how to analyze log1.txt and store the result in result.txt.

```
umdh -v log1.txt  > result.txt
```

Symbol Files for Analyzing a Log File

Suppose you have two computers: a *logging computer* where you create a UMDH log and an *analysis computer* where you analyze the UMDH log. The symbol path on your analysis computer must point to the symbols for the version of Windows that was loaded on the logging computer at the time the log was made. Do not point the symbol path on the analysis computer to a symbol server. If you do, UMDH will retrieve symbols for the version of Windows that is running on the analysis computer, and UMDH will not display meaningful results.

Interpreting a Log Comparison

12/5/2018 • 2 minutes to read • Edit Online

You can generate multiple User-Mode Dump Heap (UMDH) logs of the same process over time. Then, you can use UMDH to compare the logs and determine which call stack allocations have grown the most between trials.

For example, the following command directs UMDH to compare two UMDH logs, Log1.txt and Log2.txt, and redirects the output to a third file, Compare.txt.

```
umdh -v Log1.txt Log2.txt > Compare.txt
```

The resulting Compare.txt file lists the call stacks recorded in each log and, for each stack, displays the change in heap allocations between the log files.

For example, the following line from the file shows the change in allocation size for the functions in the call stack labeled "Backtrace00053."

In Log1.txt, the calls in the stack accounts for 40,432 (0x9DF0) bytes, but in Log2.txt, the same call stack accounts for 61,712 (0xF110) bytes, a difference of 21,280 (0x5320) bytes.

```
+ 5320 (f110 - 9df0) 3a allocs BackTrace00053
Total increase == 5320
```

Following is the stack for the allocation:

```
ntdll!RtlDebugAllocateHeap+0x000000FD
ntdll!RtlAllocateHeapSlowly+0x0000005A
ntdll!RtlAllocateHeap+0x00000808
MyApp!_heap_alloc_base+0x00000069
MyApp!_heap_alloc_dbg+0x000001A2
MyApp!_nh_malloc_dbg+0x00000023
MyApp!_nh_malloc+0x00000016
MyApp!operator new+0x0000000E
MyApp!LeakyFunc+0x0000001E
MyApp!main+0x0000002C
MyApp!mainCRTStartup+0x000000FC
KERNEL32!BaseProcessStart+0x0000003D
```

An examination of the call stack shows that the **LeakyFunc** function is allocating memory by using the Visual C++ run-time library. If examination of the other log files shows that the allocation grows over time, you might be able to conclude that memory allocated from the heap is not being freed.

Symbol Files for Analyzing a Log File

Suppose you have two computers: a *logging computer* where you create a UMDH log and an *analysis computer* where you analyze the UMDH log. The symbol path on your analysis computer must point to the symbols for the version of Windows that was loaded on the logging computer at the time the log was made. Do not point the symbol path on the analysis computer to a symbol server. If you do, UMDH will retrieve symbols for the version of Windows that is running on the analysis computer, and UMDH will not display meaningful results.

See Also

Using UMDH to Find a User-Mode Memory Leak

USBView

8/5/2019 • 2 minutes to read • Edit Online

Universal Serial Bus Viewer (USBView) or USBView.exe is a Windows graphical UI app that you can use to browse all USB controllers and connected USB devices on your computer. USBView works on all versions of Windows.

Where to get USBView

To download and use USBView, complete the following steps:

1. Download and install the Windows SDK.
2. During the installation, select only the **Debugging Tools for Windows** box and clear all other boxes.
3. By default, on a x64 PC the SDK will install USBView to the following directory.

   ```
   C:\Program Files (x86)\Windows Kits\10\Debuggers\x64
   ```

4. Open the kits debugger directory for the processor type you're running, and then select USBView.exe to start the utility.

USBView source code

USBView is also available in the Windows driver samples repository on GitHub.

Use USBView

USBView can enumerate USB host controllers, USB hubs, and attached USB devices. It can also query information about the devices from the registry and through USB requests to the devices.

The main USBView window contains two panes. The left pane displays a connection-oriented tree view that you can use to select any USB device.

The right pane displays the USB data structures that pertain to the selected USB device. These structures include Device, Configuration, Interface, and Endpoint Descriptors, as well as the current device configuration.

Use Device Manager to display USB info

To use Device Manager to display USB info:

1. Select Windows logo key+R, enter **devmgmt.msc** into the pop-up box, and then select Enter.
2. In Device Manager, select your computer so that it's highlighted.
3. Select **Action**, and then select **Scan for hardware changes**.
4. Select **View**, and then select **Hidden Devices** to display additional devices (for example, those that are not currently active).
5. Expand the **Universal Serial Bus controllers** node in Device Manager and select the device in question.
6. Right-click to select **Properties** and view summary device status info.
7. Select the **Details** tab to view additional info.

8. Select **Property** to view details such as **Status** or **Problem code**.

Windows USB troubleshooter

If you're trying to diagnose a USB device that doesn't eject using the **Safely Remove Hardware** dialog box, try using the Windows USB Troubleshooter.

Tools Related to Debugging Tools for Windows

6/7/2019 • 2 minutes to read • Edit Online

This section describes tools that are related to debugging but are not included in the Debugging Tools for Windows package.

Application Verifier

Monitor application actions while the application runs, subject the application to a variety of stresses and tests, and generate a report about potential errors in application execution or design. Application Verifier is included in the Windows Software Development Kit (SDK) for Windows. You can find information about downloading and installing the Windows SDK for Windows here.

Windows Error Reporting

You can configure Windows Error Reporting (WER) to write user-mode dump files when exceptions and other errors occur in user-mode code. WER is included in Windows Vista and later versions of Windows.

Related topics

Tools Included in Debugging Tools for Windows

Debugging Tools for Windows

Windows Error Reporting

12/5/2018 • 2 minutes to read • Edit Online

Windows Error Reporting (WER) is included in Windows Vista and later versions of Windows. You can configure WER to write user-mode dump files when exceptions and other errors occur in user-mode code. For more information, see Enabling Postmortem Debugging and Collecting User-Mode Dumps.

WER replaced Dr. Watson, which was included in Windows XP.

Source Code

12/5/2018 • 2 minutes to read • Edit Online

This section contains the following topics.

- Source Path
- Using a Source Server
- Creating Your Own Source Control System
- SrcSrv

Source Path

12/5/2018 • 3 minutes to read • Edit Online

The *source path* specifies the directories where the C and C++ source files are located.

If you are debugging a user-mode process on the computer where the executable file was built, and if the source files are still in their original location, the debugger can automatically locate the source files.

In most other situations, you have to set the source path or load the individual source files.

When you are performing remote debugging through the debugger, the debugging server uses the source path. If you are using WinDbg as your debugger, each debugging client also has its own *local source path*. All source-related commands access the source files on the local computer. You have to set the proper paths on any client or server that you want to use source commands.

This multiple-path system also enables a debugging client to use source-related commands without actually sharing the source files with other clients or with the server. This system is useful if there are private or confidential source files that one of the users has access to.

You can also load source files at any time, regardless of the source path.

Source Path Syntax

The debugger's source path is a string that consists of multiple directory paths, separated by semicolons.

Relative paths are supported. However, unless you always start the debugger from the same directory, you should add a drive letter or a network share before each path. Network shares are also supported.

Note If you are connected to a corporate network, the most efficient way to access source files is to use a source server. You can use a source server by using the **srv*** string within your source path. For more information about source servers, see Using a Source Server.

Controlling the Source Path

To control the source path and local source path, you can do one of the following:

- Before you start the debugger, use the _NT_SOURCE_PATH environment variable to set the source path. If you try to add an invalid directory through this environment variable, the debugger ignores this directory.
- When you start the debugger, use the **-srcpath**command-line option to set the source path.
- Use the **.srcpath (Set Source Path)** command to display, set, change, or append to the source path. If you are using a source server, **.srcfix (Use Source Server)** is slightly easier.
- (WinDbg only) Use the **.lsrcpath (Set Local Source Path)** command to display, set, change, or append to the local source path. If you are using a source server, **.lsrcfix (Use Local Source Server)** is slightly easier. You can also use the WinDbg Command-Line with the parameter -lscrpath. For more information, see **WinDbg Command-Line Options**.
- (WinDbg only) Use the File | Source File Path command or press CTRL+P to display, set, change, or append to the source path or the local source path.

You can also directly open or close a source file by doing one of the following:

- Use the **lsf (Load or Unload Source File)** command to open or close a source file.

- (WinDbg only) Use the **.open (Open Source File)** command to open a source file.
- (WinDbg only) Use the file | open source file command or press ctrl+o to open a source file. you can also use the **open source file (ctrl+o)** button ( ) on the toolbar.

  Note When you use **File | Open Source File** (or its shortcut menu or button equivalents) to open a source file, the path of that file is automatically appended to the source path.
- (WinDbg only) Use the File | Recent Files command to open one of the four source files that you most recently opened in WinDbg.
- (WinDbg only) Use the File | Close Current Window command or click the **Close** button in the corner of the Source window to close a source file.

For more information about how to use source files, see Debugging in Source Mode.

Using a Source Server

12/5/2018 • 2 minutes to read • Edit Online

A source server enables the debugger to automatically retrieve the source files that match the current target. To use a source server, you must be debugging binaries that have been source indexed at build time and whose source file locations are embedded in the PDB files.

Debugging Tools for Windows includes the source server SrcSrv (Srcsrv.exe).

Using SrcSrv with a Debugger

SrcSrv can be used with WinDbg, KD, NTSD, or CDB.

To use SrcSrv with the debugger, enter the following command to set the source path to srv*.

```
.srcfix
```

You can get the same result by entering the following command.

```
.srcpath srv*
```

Setting the source path to srv* tells the debugger that it should retrieve source files from locations specified in the target modules' symbol files.

If you want to use SrcSrv and also include a list of directories in your source path, use semicolons to separate `srv*` from any directories that are in the path.

For example:

```
.srcpath srv*;c:\someSourceCode
```

If the source path is set as shown in the preceding example, the debugger first uses SrcSrv to retrieve source files from locations specified in the target modules' symbol files. If SrcSrv is unable to retrieve a source file, the debugger attempts to retrieve it from c:\someSourceCode. Regardless of whether srv* is the first element in the path or appears later, the debugger always uses SymSrv before it searches any other directories listed in the path.

You can also use **.srcfix+** to append `srv*` to your existing source path, as shown in the following example.

```
3: kd> .srcpath c:\mySource
Source search path is: c:\mySource
3: kd> .srcfix+
Source search path is: c:\mySource;SRV*
```

If a source file is retrieved by the source server, it will remain on your hard drive after the debugging session is over. Source files are stored locally in the src subdirectory of the home directory (unlike the symbol server, the source server does not specify a local cache in the `srv*` syntax itself). The home directory defaults to the debugger installation directory; it can be changed by using the **!homedir** extension or by setting the DBGHELP_HOMEDIR environment variable. If this subdirectory does not already exist, it will be created.

If you use the **.open (Open Source File)** command to open a new source file through SrcSrv, you must include the -m Address parameter.

For information on how to index your sources, or if you plan to create your own source control provider module, see SrcSrv.

Using AgeStore to Reduce the Cache Size

Any source files downloaded by SrcSrv will remain on your hard drive after the debugging session is over. To control the size of the source cache, the AgeStore tool can be used to delete cached files that are older than a specified date, or to reduce the contents of the cache below a specified size. For details, see AgeStore.

SrcSrv

12/5/2018 • 2 minutes to read • Edit Online

The SrcSrv tool (Srcsrv.dll) enables a client to retrieve the exact version of the source files that were used to build an application. Because the source code for a module can change between versions and over the course of years, it is important to look at the source code as it existed when the version of the module in question was built.

SrcSrv retrieves the appropriate files from source control. To use SrcSrv, the application must have been source-indexed.

This section includes:

Using SrcSrv

Source Indexing

Source Control Systems

HTTP Sites and UNC Shares

Using SrcSrv

12/5/2018 • 2 minutes to read • Edit Online

To use SrcSrv with WinDbg, KD, NTSD, or CDB, verify that you have installed a recent version of the Debugging Tools for Windows package (version 6.3 or later). Then, include the text `srv*` in the source path, separated by semicolons from any directories that are also in the source path.

For example:

```
.srcpath srv*;c:\someSourceCode
```

If the source path is set as shown in the preceding example, the debugger first uses SrcSrv to retrieve source files from locations specified in the target modules' symbol files. If SrcSrv is unable to retrieve a source file, the debugger attempts to retrieve it from c:\someSourceCode. Regardless of whether srv* is the first element in the path or appears later, the debugger always uses SymSrv before it searches any other directories listed in the path.

If a source file is retrieved by SrcSrv, it remains on your hard drive after the debugging session is over. Source files are stored locally in the src subdirectory of the home directory (unlike the symbol server, the source server does not specify a local cache in the `srv*` syntax itself). The home directory defaults to the Debugging Tools for Windows installation directory; it can be changed by using the **!homedir** extension or by setting the DBGHELP_HOMEDIR environment variable. If the src subdirectory of the home directory does not already exist, it is created.

Debugging SrcSrv

If you experience any trouble extracting the source files from the debugger, start the debugger with the -n command-line parameter to view the actual source extraction commands along with the output of those commands. The !sym noisy command does the same thing, but you may have already missed important information from previous extraction attempts. This is because the debugger gives up trying to access source from version control repositories that appear to be unreachable.

Retrieving Source Files

If you use the **.open (Open Source File)** command to open a new source file through SrcSrv, you must include the -m Address parameter.

To facilitate the use of SrcSrv from tools other than the debuggers listed previously, the DbgHelp API provides access to SrcSrv functionality through the **SymGetSourceFile** function. To retrieve the name of the source file to be retrieved, call the **SymEnumSourceFiles** or **SymGetLineFromAddr64** function. For more details on the DbgHelp API, see the dbghelp.chm documentation, which can be found in the sdk/help subdirectory of the Debugging Tools for Windows installation directory, or see Debug Help Library.

Using AgeStore to Reduce the Cache Size

Any source files downloaded by SrcSrv remain on your hard drive after the debugging session is over. To control the size of the source cache, the AgeStore tool can be used to delete cached files that are older than a specified date or to reduce the contents of the cache below a specified size. For details, see AgeStore.

Source Indexing

12/5/2018 • 2 minutes to read • Edit Online

Generally, binaries are source-indexed during the build process after the application has been built. The information needed by SrcSrv is stored in the .pdb files. SrcSrv currently supports the following source control systems:

- Perforce
- Microsoft Visual SourceSafe
- Microsoft Team Foundation Server

You can also create a custom script to index your code for a different source control system. One such module for Subversion is included in this package.

SrcSrv includes five tools that are used in the source indexing process:

The Srcsrv.ini File

The Ssindex.cmd Script

The SrcTool Utility

The PDBStr Tool

The VSSDump Tool

These tools are installed with Debugging Tools for Windows in the subdirectory srcsrv, and should remain installed in the same path as SrcSrv. The PERL scripts in these tools require PERL 5.6 or later.

The Srcsrv.ini File

12/5/2018 • 2 minutes to read • Edit Online

The Srcsrv.ini file is the master list of all source control servers. Each entry has the following format:

```
MYSERVER=ServerInfo
```

When using Perforce, the *ServerInfo* consists of the full network path to the server, followed by a colon, followed by the port number it uses. For example:

```
MYSERVER=machine.corp.company.com:1666
```

Srcsrv.ini is a required file when you are actually source indexing a build using the modules shipped with this package. This entry creates an alias that is used to describe the server info. The value should be unique for every server that you support.

This file can also be installed on the computer running the debugger. When SrcSrv starts, it looks at Srcsrv.ini for values; these values override the information contained in the .pdb file. This enables users to configure a debugger to use an alternative source control server at debug time. However, if you manage your servers well and do not rename them, there should be no need to include this file with your client debugger installations.

This file also serves other purposes on the client side. For more information, see the sample Srcsrv.ini file installed with SrcSrv tools.

Using a Different Location or File Name

By default, SrcSrv uses as its master configuration file the file named Srcsrv.ini, located in the srcsrv subdirectory of the Debugging Tools for Windows installation directory.

You can specify a different file for configuration by setting the SRCSRV_INI_FILE environment variable equal to the full path and file name of the desired file.

For example, if several people want to share a single configuration file, they could place it on a share accessible to all of their systems, and then set an environment variable like the following:

```
set SRCSRV_INI_FILE=\\ourserver\ourshare\bestfile.txt
```

The Ssindex.cmd Script

12/5/2018 • 2 minutes to read • Edit Online

The Ssindex.cmd script builds the list of files checked into source control along with the version information of each file. It stores a subset of this information in the .pdb files generated when you built the application. SSIndex uses one of the following Perl modules to interface with source control:

- p4.pm (Perforce)
- vss.pm (Visual SourceSafe)
- tfs.pm (Team Foundation Servers)
- svn.pm (Subversion)

For more information, run Ssindex.cmd with the **-?** or **-??** (verbose help) option or examine the script.

The SrcTool Utility

12/5/2018 • 2 minutes to read • Edit Online

The SrcTool (Srctool.exe) utility lists all files indexed within the .pdb file. For each file, it lists the full path, source control server, and version number of the file. You can use this information for reference.

You can also use SrcTool to list the raw source file information from the .pdb file. To do this, use the **-s** switch on the command line.

SrcTool has other options as well. Use the **?** switch to see them. Of most interest is that this utility can be used to extract all of the source files from version control. This is done with the **-x** switch.

Note Previous versions of this program created a directory called src below the current directory when extracting files. This is no longer the case. If you want the src directory used, you must create it yourself and run the command from that directory.

The PDBStr Tool

12/5/2018 • 2 minutes to read • Edit Online

The PDBStr (Pdbstr.exe) tool is used by the indexing scripts to insert the version control information into the "srcsrv" alternative stream of the target .pdb file. It can also read any stream from a .pdb file. You can use this information to verify that the indexing scripts are working properly.

For more information, run PDBStr with the **-?** option.

The VSSDump Tool

12/5/2018 • 4 minutes to read • Edit Online

The VSSDump (Vssdump.exe) tool is used by the indexing script for Microsoft Visual SourceSafe. It gathers the list of source files to be indexed. This program is also a valuable command-line utility that you can use to examine which files may be processed by the indexing script.

To prepare for source indexing, edit Srcsrv.ini so that it contains an entry for your version control server. This is an operation that you must do only once. Details are listed in the sample version Srcsrv.ini. You can use an environment variable or switch to the indexing script to denote the location of this file. However, it is best to put it in the same directory as the script because the contents of this file are intended to be global across all projects in your business or development system. This file serves to uniquely identify different version control servers. Note that you can provide a different version of this file to debuggers so that they look at a replicated copy of the version control server, which can be useful if you want to reduce traffic on the server.

To source-index a particular build, make certain that no source files are checked out on the build computer. If any files are checked out and edited, those changes are not reflected in the final source indexed .pdb files. Furthermore, if your build process includes a pre-compilation pass that generates source files from other files, you must check in those generated files to version control as part of the pre-compilation.

Upon completion of the build, set the current working directory to be the root of all source files and generated .pdb files. Then run SSIndex. You must specify what version control system you are using as a parameter. For example:

ssindex.cmd -server=vss

SSIndex accepts parameters that allow you to run the script from anywhere and to specify the location of the source files and .pdb files separately. This is most useful if the source is kept in another location from the output .pdb files. For example:

ssindex.cmd -server=vss -source=c:*source* **-symbols=c:***outputdir*

These directories can also be specified with environment variables. Use the -? or -?? command line options for more information.

Here is an example of the output generated by this script:

```
## C:\ >ssindex.cmd -system=vss -?

SSIndex.cmd [/option=<value> [...]] [ModuleOptions] [/Debug]
General SrcSrv settings:
     NAME              SWITCH      ENV. VAR        Default
  ------------------------------------------------------------------------------
  1) srcsrv.ini        Ini         SRCSRV_INI      .\srcsrv.ini
  2) Source root       Source      SRCSRV_SOURCE   .
  3) Symbols root      Symbols     SRCSRV_SYMBOLS  .
  4) Control system    System      SRCSRV_SYSTEM   <N/A>
  5) Save File (opt.)  Save        SRCSRV_SAVE     <N/A>
  6) Load File (opt.)  Load        SRCSRV_LOAD     <N/A>
Visual Source Safe specific settings:
     NAME            SWITCH      ENV. VAR        Default
  ------------------------------------------------------------------------------
  A) VSS Server      Server      SSDIR            <N/A>
  B) VSS Client      Client      SSROOT           <Current directory>
  C) VSS Project     Project     SSPROJECT        <N/A>
  D) VSS Label       Label       SSLABEL          <N/A>
Precedence is: Default, environment, cmdline switch. (ie. env overrides default,
switch overrides env).
Using '/debug' will turn on verbose output.
Use "SSIndex.cmd -??" for verbose help information.
## See SrcSrv documentation for more information.
```

You can also use one of the provided wrapper scripts (Vssindex.cmd) to avoid specifying the version control system. The script source-indexes all .pdb files found in the current directory and below with version control information to locate all source files found in the current directory and below. You can specify different locations for these files by using environment variables and command-line switches.

Upon completion of the source indexing, you can test the output by running SrcTool on the .pdb files. This program displays data that indicates whether or not the .pdb file is source indexed. It also displays the specific information for every source file. Lastly, it displays the number of indexed sources and the number of sources that no indexing information was found for. It sets an %ERRORLEVEL% of -1 if the file is not source-indexed. Otherwise, it sets %ERRORLEVEL% to the number of indexed source files.

VSSDump can also be used independently to diagnose issues while source-indexing. The syntax is as follows:

vssdump.exe *Options*

Options can be any combination of the following options.

-a
Causes all projects to be searched, rather than restricting to the current project. This option may not be used with the **-p** option.

-p *ProjectName*
Causes *VSSDump* to restrict its search to the project specified by *ProjectName*. If this option is not present, the current project is used. This option may not be used with the **-a** option.

-d *RootDirectory*
Causes *VSSDump* to restrict its search to the root directory specified by *RootDirectory*. If this option is not present, the current directory is used.

-l *Label*
Causes *VSSDump* to list only those files with a label that matches that specified by *Label*.

VSSDump-**v** *SharePath*
Specifies that the location of the Virtual SourceSafe database is in *SharePath*. This option overrides the path specified in the SSDIR environment variable.

-r
Causes *VSSDump* to search subdirectories recursively.

-f
Causes *VSSDump* to list directories containing source files without listing the files themselves.

-i
Causes *VSSDump* to ignore the current directory and list the entire project. This option may not be used with **-r**.

-s
Causes *VSSDump* to format output for use with script files.

-c
Causes *VSSDump* to display only the Virtual SourceSafe configuration information.

Source Control Systems

12/5/2018 • 2 minutes to read • Edit Online

SrcSrv includes provider modules for Visual SourceSafe (vss.pm), Perforce (p4.pm), Team Foundation Servers (tfn.pm), and Subversion (svn.pm). There is also a Concurrent Versions Systems (CVS) module for use with SrcSrv, but it has not been tested extensively. It is also possible to generate your own modules to support other versions of control systems.

This section includes:

Using Visual SourceSafe

Debugging with Visual SourceSafe

Using CVS

Using Other Source Control Systems

Using Visual SourceSafe

12/5/2018 • 2 minutes to read • Edit Online

Visual SourceSafe is an x86-based application, and the source indexing scripts and tools associated with it are installed only with the x86 package of Debugging Tools for Windows.

To successfully use Visual SourceSafe with SrcSrv, several defaults must be set in the system, including the Visual SourceSafe default database, the current project, and the working folder. You must also stamp each build project with a label so that Visual SourceSafe can differentiate between versions of a given file. Finally, you must set up any credentials needed to access the Visual SourceSafe database.

Visual SourceSafe Database

The default Visual SourceSafe database can be set with the SSDIR environment variable. If it is not set there, the SrcSrv indexing script can usually determine this from the registry. If it cannot, the script prompts you to set SSDIR. Regardless, this database value must be listed in your Srcsrv.ini file along with a simple alias to identify the database. More instructions can be found in the comments in Srcsrv.ini. You should add the entry in the variables section of Srcsrv.ini using the following syntax:

```
MYDATABASE=\\server\VssShare
```

Current Project

Visual SourceSafe uses the concept of a *current project*. The SrcSrv indexing script respects this value and limits processing to those source files that are part of the current project. To display the current project, use the following command:

```
ss.exe project
```

To change the current project, use the following command:

```
ss.exe cp Project
```

where *Project* indicates the current project. For example, to process all projects, set the current project to the root:

```
ss.exe cp $/
```

Working Folder

Visual SourceSafe projects are associated with *working folders*. A working folder is the location on the client computer that corresponds to the root of a project. However it is possible for such a computer to obtain and build source without a working folder being specified, usually when creating projects through the Visual Studio interface or by using the command:

```
ss.exe get -R
```

However, this mode of operation is incompatible with SrcSrv indexing. SrcSrv requires that a working folder be specified. To set the working folder, use the command:

```
ss.exe workfold Project Folder
```

where *Project* indicates the current project and *Folder* indicates the location corresponding to the root of the project.

Labels

Visual SourceSafe cannot determine what version of a file exists within the working directory of a build machine. Consequently, you must use labels to stamp the project with an identifier that is used to extract source files on the debugger client. Thus, before indexing, you must verify that all changes are checked into the database and then apply a label to the project. Labels can be applied using the command:

```
ss.exe label Project
```

where *Project* indicates the current project.

In the following example, the label "VERSION_3" is applied to a project called "$/sdktools":

```
E:\nt\user>ss.exe label $/sdktools
 Label for $/sdktools: VERSION_3
 Comment for $/sdktools:
 This is a comment.
```

After the label is applied, you can specify this label to the SrcSrv indexing script by setting the environment variable, SSLABEL, or by passing it as a command-line parameter, as in the following example:

```
vssindex.cmd -label=VERSION_3
```

Again, this label is required for SrcSrv indexing to work. Note that if you pass a label that does not exist in the project, it can take a long time for the script to complete and the results are of no use.

Credentials

If your Visual SourceSafe database requires a user and optional password for access, these values must be set through the SSUSER and SSPWD environment variables. This applies not only at the time the build is indexed, but also on the debugger client.

Debugging with Visual SourceSafe

12/5/2018 • 2 minutes to read • Edit Online

In order for source files indexed using Visual SourceSafe to work properly with the debugger, you must configure your system properly.

If the Visual SourceSafe database requires a user and optional password for access, these values must be set using the SSUSER and SSPWD environment variables. Furthermore, the version of SrcSrv that ships with Visual Studio 2005 cannot detect whether Visual SourceSafe is prompting for credentials, which causes the application to stop responding.. Upgrade to the version of SrcSrv that ships with Debugging Tools for Windows to prevent this.

If Visual SourceSafe is not set in the path of your debugging computer, you can get around this by adding an entry to the Srcsrv.ini file that works with your debugger. When using a standard installation of Visual Studio, this file should be located in

```
%PROGRAMFILES%\Microsoft Visual Studio 8\Common7\IDE\srcsrv.ini
```

In the [trusted commands] section of Srcsrv.ini, add an entry of the form

```
ss.exe="Path"
```

where *Path* is the location of Ss.exe.

Using CVS

12/5/2018 • 2 minutes to read • Edit Online

The CVS module for Source Server was developed using Concurrent Versions System (CVS) 1.11.17 (client). It has not been tested with any other versions of CVS. Furthermore, the current version of the module is a beta version.

CVSROOT

On the computer on which you source index the build, CVSROOT cannot contain password and user information. Use cvs.exe to set your credentialing information.

To prepare the Srcsrv.ini file for CVS indexing you must enter an alias for your repository that uniquely distinguishes it from any others in your network. This repository must match the value of CVSROOT in your environment. There is no need to set this value in the copy of Srcsrv.ini that you keep with your debugger clients because the alias is defined in the source indexed .pdb file.

Client Computer

The client computer that extracts files during debugging does not need a CVS sandbox or CVSROOT set. It does need CVS binaries in the path, and if the repository is locked, you must set the username and password with Cvs.exe.

Revision Tags

CVS is unable to extract a file by its version number. Instead, it must be done using what is known as a *tag*. When indexing a CVS-based system, you must ensure that all changes are checked into the repository and then apply a tag using the "cvs tag" command. Then, when indexing the file, make certain you use the "label" command-line parameter to specify the tag that you want to associate with the build you are indexing. You can achieve the same result by setting CVS_LABEL in the environment. Other values can be set from the environment or the command line. Use the **-??** command-line option with SSIndex to examine your choices and to verify that all was configured correctly:

```
ssindex.cmd -system=cvs -??
```

Using Other Source Control Systems

12/5/2018 • 2 minutes to read • Edit Online

Included in this package is Svn.pm. A closer look at this file shows how you can modify one of the existing scripts to support the Subversion Version Control system. Other control systems, even a control system that you create yourself, can be supported by creating your own provider module.

This section includes:

Creating Your Own Provider Module

Creating Your Own Source Control System

Creating Your Own Provider Module

12/5/2018 • 2 minutes to read • Edit Online

In general, to create your own provider module, you must implement the following set of interfaces.

$module::SimpleUsage()

Purpose
Displays simple module usage information to STDOUT.

Parameters
None

Return Value
None

$module::VerboseUsage()

Purpose
Displays in-depth module usage information to STDOUT.

Parameters
None

Return Value
None

$objref = $module::new(*@CommandArguments***)**

Purpose
Initializes an instance of the provider module.

Parameters

@CommandArguments
All @ARGV arguments that are not recognized by ssindex.cmd as being general arguments.

Return Value
A reference that can be used in later operations.

$objref->GatherFileInformation(***$SourcePath*,***$ServerHashReference***)**

Purpose
Enables the module to gather the required source-indexing information for the directory specified by the *$SourcePath* parameter. The module should not assume that this entry is called only once for each object instancebecause SSIndex may call it multiple times for different paths.

Parameters

$SourcePath
The local directory containing the source to be indexed.

$ServerHashReference
A reference to a hash containing all of the entries from the specified Srcsrv.ini file.

Return Value
None

(*$VariableHashReference*,*$FileEntry***) = $objref**->**GetFileInfo(***$LocalFile***)**

Purpose
Provides the necessary information to extract a single, specific file from the source control system.

Parameters

$LocalFile
A fully qualified file name.

Return Values

$VariableHashReference
A hash reference of the variables necessary to interpret the returned *$FileEntry*. Ssindex.cmd caches these variables for every source file used by a single debug file to reduce the amount of information written to the source index stream.

$FileEntry
The file entry to be written to the source index stream to allow SrcSrv to extract this file from source control. The exact format of this line is specific to the source control system.

*$TextString***= $objref->LongName()**

Purpose
Provides a descriptive string to identify the source control system to the end user.

Parameters
None

Return Value

$TextString
The descriptive name of the source control system.

@StreamVariableLines=$objref->SourceStreamVariables()

Purpose
Enables the source control system to add source-control-specific variables to the source stream for each debug file. The sample modules use this method for writing the required EXTRACT_CMD and EXTRACT_TARGET variables.

Parameters
None

Return Value

@StreamVariableLines
The list of entries for the source stream variables.

Creating Your Own Source Control System

12/5/2018 • 2 minutes to read • Edit Online

This section enables you to understand how to prepare instrumentation scripts so that SrcSrv can be integrated into your build or version control system. The implementation is dependent on the language specification version that ships with SrcSrv.

Many of the specifics of how Srcsrv.dll is called by symbol handler code are discussed. However, this information is not static and will not become so in the future. No one should attempt to write code that calls Srcsrv.dll directly. This is reserved for DbgHelp or the Visual Studio products. Third-party vendors wanting to integrate such functionality into their products should use the **SymGetSourceFile** function. This function, along with others in the DbgHelp API, is described in the DbgHelp documentation, which can be found in the sdk/help subdirectory of the Debugging Tools for Windows installation directory.

This section includes:

Language Specification 1

Language Specification 2

Language Specification 1

12/5/2018 • 6 minutes to read • Edit Online

The first version of SrcSrv works as follows. (This behavior may change in future versions.)

First, the client calls **SrcSrvInit** with the target path to be used as a base for all source file extractions. This path is stored in the special variable TARG.

When DbgHelp loads a module's .pdb file, it extracts the SrcSrv stream from the .pdb file and passes this data block to SrcSrv by calling **SrcSrvLoadModule**.

Then when DbgHelp needs to obtain a source file, it calls **SrcSrvGetFile** to retrieve a specified source file from version control.

SrcSrv reviews all the source file entries in the data block for an entry that matches exactly the requested source specification. This match is found in VAR1.

After SrcSrv finds the entry, it fills in the special variables (VAR1, VAR2, etc.) with the contents of this source file entry. Then the SRCSRVTRG variable is resolved using these special variables.

The following shows how the SRCSRVTRG variable is resolved using the special variables. We assume that the source path is still:

```
c:\proj\src\file.cpp*TOOLS_PRJ*tools/mytool/src/file.cpp*3
```

Each line shows the resolution of one more special variable. The resolved variables are bold.

```
SRCSRVTRG=%sdtrg%
SDTRG=%targ%\%var2%\%fnbksl%(%var3%)\%var4%\%fnfile%(%var1%)
c:\src\%var2%\%fnbksl%(%var3%)\%var4%\%fnfile%(%var1%)
c:\src\WIN_SDKTOOLS\%fnbksl%(%var3%)\%var4%\%fnfile%(%var1%)
c:\src\WIN_SDKTOOLS\%fnbksl%( sdktools/debuggers/srcsrv/shell.cpp )\%var4%\%fnfile%(%var1%)
c:\src\WIN_SDKTOOLS\ sdktools\debuggers\srcsrv\shell.cpp\%var4%\%fnfile%(%var1%)
c:\src\WIN_SDKTOOLS\sdktools\debuggers\srcsrv\shell.cpp\3\%fnfile%(%var1%)
c:\src\WIN_SDKTOOLS\sdktools\debuggers\srcsrv\shell.cpp\3\%fnfile%( c:\db\srcsrv\shell.cpp)
c:\src\WIN_SDKTOOLS\sdktools\debuggers\srcsrv\shell.cpp\3\shell.cpp
```

Notice how this generated target path is unique and does not allow two versions of the same file to be extracted to the same location.

SrcSrv now looks to see if the file is already there. If it is, SrcSrv returns the location to the caller. Otherwise, SrcSrv builds the execution command to extract the file by resolving SRCSRVCMD.

In the following example, each line shows the resolution of one more special variable. The resolved variables are bold.

```
DEPOT=//depot
WIN_SDKTOOLS= sserver.microsoft.com:4444
SRCSRVCMD=%sdcmd%
SDCMD=sd.exe -p %fnvar%(%var2%) print -o %srcsrvtrg% -q %depot%/%var3%#%var4%
sd.exe -p %fnvar%(WIN_SDKTOOLS) print -o %srcsrvtrg% -q %depot%/%var3%#%var4%
sd.exe -p sserver.microsoft.com:4444  print -o %srcsrvtrg% -q %depot%/%var3%#%var4%
sd.exe -p sserver.microsoft.com:4444  print -o
c:\src\WIN_SDKTOOLS\sdktools\debuggers\srcsrv\shell.cpp\3\shell.cpp -q %depot%/%var3%#%var4%
sd.exe -p sserver.microsoft.com:4444  print -o
c:\src\WIN_SDKTOOLS\sdktools\debuggers\srcsrv\shell.cpp\3\shell.cpp -q //depot/%var3%#%var4%
sd.exe -p sserver.microsoft.com:4444  print -o
c:\src\WIN_SDKTOOLS\sdktools\debuggers\srcsrv\shell.cpp\3\shell.cpp -q //depot/
sdktools/debuggers/srcsrv/shell.cpp#%var4%
sd.exe -p sserver.microsoft.com:4444  print -o
c:\src\WIN_SDKTOOLS\sdktools\debuggers\srcsrv\shell.cpp\3\shell.cpp -q //depot/
sdktools/debuggers/srcsrv/shell.cpp#3
```

Now SrcSrv executes this command. If the result of this command is a file in the expected location, this path is returned to the caller.

Note that if a variable cannot be resolved, an attempt is made to look it up as an OS environment variable. If that fails, the variable name is deleted from the text being processed.

Two consecutive percent sign characters are interpreted as a single percent sign.

Source Server Data Blocks

SrcSrv relies on two blocks of data within the .pdb file, the source file list and the data block.

The source file list is created automatically when a module is built. This list contains fully qualified paths to the source files used to build the module.

The data block is created during source indexing. At this time, an alternative stream named "srcsrv" is added to the .pdb file. The script that inserts this data is dependent on the specific build process and source control system in use.

The data block is divided into three sections: ini, variables, and source files. The data block has the following syntax.

```
SRCSRV: ini ------------------------------------------------
VERSION=1
VERCTRL=<source_control_str>
DATETIME=<date_time_str>
SRCSRV: variables ------------------------------------------
SRCSRVTRG=%sdtrg%
SRCSRVCMD=%sdcmd%
SRCSRVENV=var1=string1\bvar2=string2
DEPOT=//depot
SDCMD=sd.exe -p %fnvar%(%var2%) print -o %srcsrvtrg% -q %depot%/%var3%#%var4%
SDTRG=%targ%\%var2%\%fnbksl%(%var3%)\%var4%\%fnfile%(%var1%)
WIN_SDKTOOLS= sserver.microsoft.com:4444
SRCSRV: source files ---------------------------------------
<path1>*<var2>*<var3>*<var4>
<path2>*<var2>*<var3>*<var4>
<path3>*<var2>*<var3>*<var4>
<path4>*<var2>*<var3>*<var4>
SRCSRV: end ------------------------------------------------
```

All text is interpreted literally, except for text enclosed in percent signs (%). Text enclosed in percent signs is treated as a variable name to be resolved recursively, unless it is one of the following functions:

%fnvar%()

The parameter text should be enclosed in percent signs and treated as a variable to be resolved.

%fnbksl%()
All forward slashes (/) in the parameter text should be replaced with backward slashes ().

%fnfile%()
All path information in the parameter text should be stripped out, leaving only the file name.

The [ini] section of the data block contains variables that describe the requirements. The indexing script can add any number of variables to this section. The following are examples:

VERSION
The language specification version. This variable is required. If you develop a script based on the current language specification, set this value to 1. The SrcSrv client code does not attempt to execute any script that has a value greater than its own. Current versions of SrcSrv use a value of 2.

VERCTL
A string that describes the source version control system. This variable is optional.

DATETIME
A string that indicates the date and time the .pdb file was processed. This variable is optional.

The [variables] section of the data block contains variables that describe how to extract a file from source control. It can also be used to define commonly used text as variables to reduce the size of the data block.

SRCSRV
Describes how to build the target path for the extracted file. This is a required variable.

SRCSRVCMD
Describes how to build the command to extract the file from source control. This includes the name of the executable file and its command-line parameters. This is required if any extraction command must be executed.

SRCSRVENV
Lists environment variables to be created during the file extraction. This is a string. Separate multiple entries with a backspace character (\b). This is an optional variable.

SRCSRVVERCTRL
Specifies the version control system in use. For Perforce, this is perforce. For Visual SourceSafe, this is vss. For Team Foundation Server, this is tfs. This variable is used to persist server errors. This is an optional variable.

SRCSRVVERRDESC
Specifies the text to display when the version control client is unable to contact the server that contains the source files to extract. SrcSrv uses this value to check for connection problems. This is an optional variable.

SRCSRVERRVAR
Indicates which variable in a file entry corresponds to a version control server. It is used by SrcSrv to identify commands that do not work, based on previous failures. The format of the text is **var***X* where *X* is the number of the variable being indicated. This is an optional variable.

The [source files] section of the data block contains an entry for each source file that has been indexed. The contents of each line are interpreted as variables with the names VAR1, VAR2, VAR3, and so on until VAR10. The variables are separated by asterisks. VAR1 must specify the fully qualified path to the source file as listed elsewhere in the .pdb file. For example:

```
c:\proj\src\file.cpp*TOOLS_PRJ*tools/mytool/src/file.cpp*3
```

is interpreted as follows:

```
VAR1=c:\proj\src\file.cpp
VAR2=TOOLS_PRJ
VAR3=tools/mytool/src/file.cpp
VAR4=3
```

In this example, VAR4 is a revision number. However, most source control systems support labeling files in such a way that the source state for a given build can be restored. Therefore, you could instead use the label for the build. The sample data block could be modified to contain a variable such as the following:

```
LABEL=BUILD47
```

Then, presuming the source control system uses the at sign (@) to indicate a label, you could modify the SRCSRVCMD variable as follows:

```
sd.exe -p %fnvar%(%var2%) print -o %srcsrvtrg% -q %depot%/%var3%@%label%
```

Handling Server Errors

Sometimes a client is unable to extract any files at all from a single version control server. This can be because the server is down and off the network or because the user does not have appropriate privileges to access the source. However, the time consumed attempting to get this source can slow things down significantly. In this situation, it is best to disable any attempt to extract from a source that has been proven to be unavailable.

Whenever SrcSrv fails to extract a file, it examines the output text produced by the command. If any part of this command contains an exact match for the contents of the SRCSRVERRDESC, all future commands to the same version control server are skipped. Note that you can define multiple error strings by adding numbers or arbitrary text to the end of the SRCSRVERRDESC variable name. Here is an example:

```
SRCSRVERRDESC=lime: server not found
SRCSRVERRDESC_2=pineapple: network error
```

The identity of the server is acquired from SRCSRVERRVAR. So if SRCSRVERRVAR contains "var2" and the file entry in the .pdb file looks like this:

```
c:\proj\src\file.cpp*TOOLS_PRJ*tools/mytool/src/file.cpp*3
```

all future attempts to obtain source using a file entry that contains "TOOLS_PRJ" in variable 2 are bypassed.

You can also add error indicators on the debugger client by editing Srcsrv.ini. See the included sample version of srcsrv.ini for details.

Language Specification 2

12/5/2018 • 2 minutes to read • Edit Online

This package ships with a srcsrv.dll that can operate without SRCSRVCMD being defined in the srcsrv data stream of a .pdb file. While such capability is of no use for normal file extraction from source control systems, it is useful for direct access of files from a UNC share or HTTP site. See HTTP Sites and UNC Shares for more details.

HTTP Sites and UNC Shares

12/5/2018 • 2 minutes to read • Edit Online

It is possible to set up a Web site that provides version-specific source to WinDbg using SrcSrv. Such a mechanism does not provide dynamic extraction of the source files from version control, but it is a valuable feature because it allows you to set the source path of WinDbg to a single unified path that provides source from many versions of many modules, instead of having to set separate paths for each debugging scenario. This is not of interest to debugging clients that have direct access to the actual version control systems but can be of assistance to those wanting to provide secure HTTP-based access to source from remote locations. The Web sites in question can be secured through HTTPS and smart cards, if desired. This same technique can be used to provide source files through a simple UNC share.

This section includes:

Setting Up the Web Site

Extracting Source Files

Modifying the Source Indexing Streams in a .pdb File

Using UNC Shares

Using HTTP Sites and UNC Shares in Conjuction with Regular Version Control

Setting Up the Web Site

10/17/2019 • 2 minutes to read • Edit Online

Set up a Web site from which to share the source files and not the root directory of the site. Your source is then available from a site such as:

```
https://SrcMachineName/source
```

In order to make your source files accessible over the Internet, you must configure the directories containing the source files.

Begin by selecting the directory in which your indexed source resides. In our examples, we call this directory c:\source and the name of the server on the network \SrcMachineName. Permissions must be set so that users can access the site, and you must add the security groups that must access the source content over the network. The amount of security to enable varies from environment to environment. For some installations, this group is **Everyone**.

To set up the permissions for the directory:

1. Open **Windows Explorer**.
2. Expand **My Computer**.
3. Expand the C: drive.
4. Right-click **c:\source** and choose **Sharing and Security**.
5. Check the **Share this folder** button.
6. Click the **Permissions** button.
7. Verify that the desired security groups have read access by adding them under **Group or user names** and checking the appropriate box.
8. Click **OK** to exit Permissions.
9. Click **OK** to exit Source Properties.

The source directory can now be used for debugging by another computer with a source path of srv*\\SrcMachineName\source.

Extracting Source Files

12/5/2018 • 2 minutes to read • Edit Online

To extract all of the source files from all the modules for which you want to provide source, use the command:

```
srctool.exe -x
```

This tool must be executed on .pdb files that have already been source-indexed for your version control system on a computer that has version control access. This places all source files into a common directory tree. Copy the contents of this tree to the root of the Web site. You can do this as often as you want on any new products or modules that you want to add. There is no worry about files overwriting each other because the directory tree structure keeps dissimilar files separated and uniquely accessible.

Walk

The Walk (Walk.cmd) script is included in Debugging Tools for Windows. This script searches recursively through a directory tree and executes any specified command on any file that matches a specified file mask. The syntax is:

```
walk.cmd FileMask Command
```

where *FileMask* specifies a file mask, with or without an accompanying starting directory, and *Command* specifies the command to be executed.

Here is an example that runs the srctool.exe file extraction command on all .pdb files in c:\symbols and its subdirectories:

```
walk.cmd c:\symbols\*.pdb srctool.exe -x
```

Modifying the Source Indexing Streams in a .pdb File

12/5/2018 • 2 minutes to read • Edit Online

For the debugger clients to use the SrcSrv Web site, the .pdb files must be modified to point to it. To do this manually, you make a copy of all the .pdb files, change them, and make them available from a separate location--usually the Web site itself.

Debugging Tools for Windows provides three files to assist in reconfiguring the .pdb files. The Cv2http.cmd and Cv2http.pl files extract the SrcSrv stream, modify it using a Perl script, and put the altered stream back in the .pdb file. The syntax is as follows:

```
cv2http.cmd PDB Alias URL
```

where *PDB* specifies the name of the .pdbfile to modify, *Alias* specifies the logical name to apply to your Web site, and *URL* specifies the full URL of the site. Note that the *Alias* parameter is stored in the PDB as a variable name that can be overridden on the debugger client in Scrsrv.ini, should you ever move the location of the Web site.

This script requires that all the standard SrcSrv tools be available in the path because it calls both SrcTool and PDBStr. Remember that Cv2http.pl is a Perl script and can be modified to meet your needs.

The third file, the Walk (walk.cmd) script, modifies an entire set of .pdb files. For example:

```
walk.cmd *.pdb cv2http.cmd HttpAlias https:///source
```

The preceding command calls Cv2http.cmd on every .pdb file in a tree, using HttpAlias for the alias and https://server/source for the URL. For more details on Walk, see Extracting Source Files.

After this command is executed on a tree of .pdb files, they are ready for installation into the Web site or whatever location in which you want to put them. Remember that you can use SrcTool and PDBStr to examine the changes to the .pdb files.

Using UNC Shares

12/5/2018 • 2 minutes to read • Edit Online

The Cv2http.cmd, Cv2http.pl, and Walk (Walk.cmd) scripts are used to provide source files from a simple UNC share. The files Cv2http.cmd and Cv2http.pl extract the SrcSrv stream, modify it using a Perl script, and put the altered stream back in the .pdb file. The syntax is as follows:

```
cv2http.cmd PDB Alias SourceRoot
```

where *PDB* specifies the name of the .pdbfile to modify, *Alias* specifies the logical name to apply to your Web site, and *SourceRoot* specifies the root of the UNC share to which you extracted the source files. Note that the *Alias* parameter is stored in the PDB as a varaible name that can be overridden on the debugger client in Scrsrv.ini, should you ever move the location of the Web site.

This script requires that all the standard SrcSrv tools be available in the path because it calls both SrcTool and PDBStr. Remember that Cv2http.pl is a Perl script and can be modified to meet your needs.

The third file, the Walk (walk.cmd) script, modifies an entire set of .pdb files. For example:

```
walk.cmd *.pdb cv2http.cmd SourceRoot \\server\share
```

The preceding command calls Cv2http.cmd on every .pdb file in a tree, using SourceRoot for the alias and \\server\share for the UNC share. For more details on Walk, see Extracting Source Files.

After this command is executed on a tree of .pdb files, they are ready for installation into the Web site or whatever location in which you want to put them. Remember that you can use SrcTool and PDBStr to examine the changes to the .pdb files.

Using HTTP Sites and UNC Shares in Conjuction with Regular Version Control

12/5/2018 • 2 minutes to read • Edit Online

You may find that you must support your developers using the standard SrcSrv functionality that extracts files from version control but must also make source files available through a Web site or UNC share. This could happen if you have set up a test lab that does not have access to version control. It is possible to support both users using the same set of .pdb files.

First, extract the source files using SrcTool; see Extracting Source Files for details. Make the share available as either a Web site or UNC share. For the current purpose, you should not convert the .pdb files using the Cv2http.cmd script.

Now on the computers that will use the HTTP/UNC shares, edit the Srcsrv.ini file that is in the debugger directory. In the variables section of the file, add the following three statements:

```
MY_SOURCE_ROOT=\\server\share
 SRCSRVCMD=
 SRCSRVTRG=%MY_SOURCE_ROOT%\%var2%\%var3%\%var4%\%fnfile%(%var1%)
```

You should replace \\server\share with the root of the UNC share that you are providing or the URL of the Web site that contains the source files. You can also change MY_SOURCE_ROOT to be any alias you want to describe this location. With these exceptions, everything else should be entered exactly as described.

All debuggers set up in this fashion ignore the standard version control extraction instructions and instead access the source files from the location specified. Meanwhile, all debuggers without these items included in Srcsrv.ini use the normal version control mechanism to extract source files.

Security Considerations

12/5/2018 • 2 minutes to read • Edit Online

This section includes:

Security Vulnerabilities

Secure Mode

Debug Privilege

12/5/2018 • 2 minutes to read • Edit Online

The debug privilege allows someone to debug a process that they wouldn't otherwise have access to. For example, a process running as a user with the debug privilege enabled on its token can debug a service running as local system.

Debug privilege is a security policy setting that allows users to attach a debugger to a process or to the kernel. An administrator can modify a security policy for a user group to include or to remove this functionality. Developers who are debugging their own applications do not need this user privilege. Developers who are debugging system components or who are debugging remote components will need this user privilege. This user privilege provides complete access to sensitive and critical operating system components. By default, this property is enabled for users with Administrator rights. A user with Administrator privileges can enable this property for other user groups.

Modifying Debug Privilege for a Process

The following code example shows how to enable the debug privilege in your process. This enables you to debug other processes that you wouldn't have access to otherwise.

```
//
//  SetPrivilege enables/disables process token privilege.
//
BOOL SetPrivilege(HANDLE hToken, LPCTSTR lpszPrivilege, BOOL bEnablePrivilege)
{
    LUID luid;
    BOOL bRet=FALSE;

    if (LookupPrivilegeValue(NULL, lpszPrivilege, &luid))
    {
        TOKEN_PRIVILEGE tp;

        tp.PrivilegeCount=1;
        tp.Privileges[0].Luid=luid;
        tp.Privileges[0].Attributes=(bEnablePrivilege) ? SE_PRIVILEGE_ENABLED: 0;
        //
        //  Enable the privilege or disable all privileges.
        //
        if (AdjustTokenPrivileges(hToken, FALSE, &tp, NULL, (PTOKEN_PRIVILEGES)NULL, (PDWORD)NULL))
        {
            //
            //  Check to see if you have proper access.
            //  You may get "ERROR_NOT_ALL_ASSIGNED".
            //
            bRet=(GetLastError() == ERROR_SUCCESS);
        }
    }
    return bRet;
}
```

The following example shows how to use this function:

```
HANDLE hProcess=GetCurrentProcess();
HANDLE hToken;

if (OpenProcessToken(hProcess, TOKEN_ADJUST_PRIVILEGES, &hToken))
{
    SetPrivilege(hToken, SE_DEBUG_NAME, TRUE);
    CloseHandle(hToken);
}
```

Security Vulnerabilities

12/5/2018 • 2 minutes to read • Edit Online

This section includes:

Security During Kernel-Mode Debugging

Security During User-Mode Debugging

Security During Postmortem Debugging

Security During Remote Debugging

Security During Kernel-Mode Debugging

12/5/2018 • 2 minutes to read • Edit Online

Security during kernel-mode debugging is never about protecting the *target* computer. The target is completely vulnerable to the debugger -- this is the very nature of debugging.

If a debugging connection was enabled during boot, it will remain vulnerable through the debugging port until its next boot.

However, you should be concerned about security on the *host* computer. In an ideal situation, the debugger runs as an application on your host computer, but does not interact with other applications on this computer. There are three possible ways in which security problems could arise:

- If you use corrupt or destructive extension DLLs, they could cause your debugger to take unexpected actions, possibly affecting the host computer.
- It is possible that corrupt or destructive symbol files could also cause your debugger to take unexpected actions, possibly affecting the host computer.
- If you are running a remote debugging session, an unexpected client might attempt to link to your server. Or perhaps the client you are expecting might attempt to perform actions that you do not anticipate.

If you want to prevent a remote user from performing actions on your host computer, use Secure Mode.

For suggestions on how to guard against unexpected remote connections, see Security During Remote Debugging.

If you are not performing remote debugging, you should still beware of bad symbol files and extension DLLs. Do not load symbols or extensions that you distrust!

Local Kernel Debugging

Only users who have debug privileges can start a local kernel debugging session. If you are the administrator of a machine that has multiple user accounts, you should be aware that any user with these privileges can start a local kernel debugging session, effectively giving them control of all processes on the computer -- and therefore giving them access to all peripherals as well.

Security During User-Mode Debugging

12/5/2018 • 2 minutes to read • Edit Online

When a user-mode debugger is active, it can effectively control any of the processes on the computer.

There are three possible ways in which security problems could arise during user-mode debugging:

- If you use corrupt or destructive extension DLLs, they could cause your debugger to take unexpected actions, possibly affecting applications other than your target.
- It is possible that corrupt or destructive symbol files could also cause your debugger to take unexpected actions, possibly affecting applications other than your target.
- If you are running a remote debugging session, an unexpected client might attempt to link to your server. Or perhaps the client you are expecting might attempt to perform actions that you do not anticipate.

For suggestions on how to guard against unexpected remote connections, see Security During Remote Debugging. After a remote client has joined a user-mode debugging session, there is no way to restrict its access to processes on your computer.

If you are not performing remote debugging, you should still beware of bad symbol files and extension DLLs. do not load symbols or extensions that you distrust!

Security During Postmortem Debugging

12/5/2018 • 2 minutes to read • Edit Online

Only an administrator can enable postmortem debugging.

However, postmortem debugging is enabled for the entire system, not just for one user. Thus, after it has been enabled, any application crash will activate the debugger that has been chosen -- even if the current user does not have administrative privileges.

Also, a postmortem debugger inherits the same privileges as the application that crashed. Thus, if a Windows service such as CSRSS and LSASS crashes, the debugger will have very high-level privileges.

You should take this into account when choosing to enable postmortem debugging.

Security During Remote Debugging

12/5/2018 • 2 minutes to read • Edit Online

There are two ways to increase security during remote debugging: by restricting who can connect to your session and by restricting the powers of someone who does connect.

Controlling Access to the Debugging Session

If you are performing remote debugging through the debugger, or using a process server or KD connection server, any computer on your local network can attempt to attach to your debugging session.

If you are using the TCP, 1394, COM, or named pipe protocols, you can require the Debugging Client to supply a password. However, this password is not encrypted during transmission, and therefore these protocols are not secure.

If you want your Debugging Server to be secure, you must use secure sockets layer (SSL) or secure pipe (SPIPE) protocol.

If you are performing remote debugging through remote.exe, you can use the **/u** parameter to prohibit connections from unauthorized users.

Restricting the Powers of the Client

If you are setting up a kernel-mode debugging session, you can restrict the debugger's ability to interfere with the host machine by using Secure Mode.

In user mode, Secure Mode is not available. You can stop an intrusive client from issuing Microsoft MS-DOS commands and running external programs by issuing the **.noshell (Prohibit Shell Commands)** command. However, there are many other ways for a client to interfere with your computer.

Note that both Secure Mode and **.noshell** will prevent both the Debugging Client and the Debugging Server from taking certain actions. There is no way to place a restriction on the client but not on the server.

Forgotten Process Servers

When you start a process server on a remote machine, the process server runs silently.

If you perform remote debugging through this process server and then end the session, the process server continues to run.

A forgotten process server is a potential target for an attack. You should always shut down an unneeded process server. Use the Kill.exe utility or Task Manager to terminate the process server.

Secure Mode

12/5/2018 • 2 minutes to read • Edit Online

When you are performing kernel-mode debugging, you can run the debugger in *Secure Mode*. This prevents the debugger from affecting the host computer, yet does not significantly decrease its freedom to debug the target computer.

Secure Mode is recommended if you are going to allow remote clients to join your debugging session.

This section includes:

Features of Secure Mode

Activating Secure Mode

Features of Secure Mode

12/5/2018 • 2 minutes to read • Edit Online

When Secure Mode is active, all commands that could be used to affect the *host* computer are deactivated, and there are some restrictions on symbol servers and debugger extensions.

The specific effects of Secure Mode are as follows:

- The **.attach (Attach to Process)**, **.create (Create Process)**, **.detach (Detach from Process)**, **.abandon (Abandon Process)**, **.kill (Kill Process)**, **.tlist (List Process IDs)**, **.dump (Create Dump File)**, **.opendump (Open Dump File)**, **.writemem (Write Memory to File)**, **.netuse (Control Network Connections)**, and **.quit_lock (Prevent Accidental Quit)** commands are not available.
- The File | Attach to a Process, File | Open Executable, Debug | Detach Debuggee, Debug | Stop Debugging, File | Open Crash Dump WinDbg menu commands are not available.
- The **.shell (Command Shell)** command is not available.
- Extension DLLs must be loaded from a local disk; they cannot be loaded from UNC paths.
- Only the two standard types of extension DLLs (wdbgexts.h and dbgeng.h) are permitted. Other types of DLLs cannot be loaded as extensions.
- If you are using a symbol server, there are several restrictions. Only SymSrv (symsrv.dll) is permitted; other symbol server DLLs will not be accepted. You may not use a downstream store for your symbols, and any existing downstream store will be ignored. HTTP and HTTPS connections are not permitted.

After it has been activated, Secure Mode cannot be turned off. For more information see, Activating Secure Mode.

Activating Secure Mode

12/5/2018 • 2 minutes to read • Edit Online

Secure Mode is only available for kernel-mode debugging. It must be activated before the debugging session has begun -- either on the debugger's command line, or when the debugger is completely dormant and is not yet being used as a server.

To activate Secure Mode, use one of the following methods:

- The **-secure** command-line option
- The **.secure 1** command
- The **.sympt+0x40000** command

If you have an existing kernel debugging session and need to discover whether you are already in Secure Mode, use the **.secure** command with no arguments. This will tell you the current status of Secure Mode.

After Secure Mode has been activated, it cannot be turned off. Even ending the debugging session will not turn it off. Secure Mode persists as long as the debugger itself is running.

Processor Architecture

12/5/2018 • 2 minutes to read • Edit Online

This section includes:

The x86 Processor

The x64 Processor

The x86 Processor

12/5/2018 • 2 minutes to read • Edit Online

This section includes:

x86 Architecture

x86 Instructions

Annotated x86 Disassembly

x86 Architecture

3/6/2019 • 9 minutes to read • Edit Online

The Intel x86 processor uses complex instruction set computer (CISC) architecture, which means there is a modest number of special-purpose registers instead of large quantities of general-purpose registers. It also means that complex special-purpose instructions will predominate.

The x86 processor traces its heritage at least as far back as the 8-bit Intel 8080 processor. Many peculiarities in the x86 instruction set are due to the backward compatibility with that processor (and with its Zilog Z-80 variant).

Microsoft Win32 uses the x86 processor in *32-bit flat mode*. This documentation will focus only on the flat mode.

Registers

The x86 architecture consists of the following unprivileged integer registers.

eax	Accumulator
ebx	Base register
ecx	Counter register
edx	Data register - can be used for I/O port access and arithmetic functions
esi	Source index register
edi	Destination index register
ebp	Base pointer register
esp	Stack pointer

All integer registers are 32 bit. However, many of them have 16-bit or 8-bit subregisters.

ax	Low 16 bits of **eax**
bx	Low 16 bits of **ebx**
cx	Low 16 bits of **ecx**
dx	Low 16 bits of **edx**

si	Low 16 bits of **esi**
di	Low 16 bits of **edi**
bp	Low 16 bits of **ebp**
sp	Low 16 bits of **esp**
al	Low 8 bits of **eax**
ah	High 8 bits of **ax**
bl	Low 8 bits of **ebx**
bh	High 8 bits of **bx**
cl	Low 8 bits of **ecx**
ch	High 8 bits of **cx**
dl	Low 8 bits of **edx**
dh	High 8 bits of **dx**

Operating on a subregister affects only the subregister and none of the parts outside the subregister. For example, storing to the **ax** register leaves the high 16 bits of the **eax** register unchanged.

When using the **? (Evaluate Expression)** command, registers should be prefixed with an "at" sign (**@**). For example, you should use **? @ax** rather than **? ax**. This ensures that the debugger recognizes **ax** as a register rather than a symbol.

However, the (@) is not required in the **r (Registers)** command. For instance, **r ax=5** will always be interpreted correctly.

Two other registers are important for the processor's current state.

eip	instruction pointer
flags	flags

The instruction pointer is the address of the instruction being executed.

The flags register is a collection of single-bit flags. Many instructions alter the flags to describe the result of the instruction. These flags can then be tested by conditional jump instructions. See x86 Flags for details.

Calling Conventions

The x86 architecture has several different calling conventions. Fortunately, they all follow the same register preservation and function return rules:

- Functions must preserve all registers, except for **eax**, **ecx**, and **edx**, which can be changed across a function call, and **esp**, which must be updated according to the calling convention.
- The **eax** register receives function return values if the result is 32 bits or smaller. If the result is 64 bits, then the result is stored in the **edx:eax** pair.

The following is a list of calling conventions used on the x86 architecture:

- Win32 (**__stdcall**)

  Function parameters are passed on the stack, pushed right to left, and the callee cleans the stack.

- Native C++ method call (also known as thiscall)

  Function parameters are passed on the stack, pushed right to left, the "this" pointer is passed in the **ecx** register, and the callee cleans the stack.

- COM (**__stdcall** for C++ method calls)

  Function parameters are passed on the stack, pushed right to left, then the "this" pointer is pushed on the stack, and then the function is called. The callee cleans the stack.

- **__fastcall**

  The first two DWORD-or-smaller arguments are passed in the **ecx** and **edx** registers. The remaining parameters are passed on the stack, pushed right to left. The callee cleans the stack.

- **__cdecl**

  Function parameters are passed on the stack, pushed right to left, and the caller cleans the stack. The **__cdecl** calling convention is used for all functions with variable-length parameters.

Debugger Display of Registers and Flags

Here is a sample debugger register display:

```
eax=00000000 ebx=008b6f00 ecx=01010101 edx=ffffffff esi=00000000 edi=00465000
eip=77f9d022 esp=05cffc48 ebp=05cffc54 iopl=0         nv up ei ng nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000286
```

In user-mode debugging, you can ignore the **iopl** and the entire last line of the debugger display.

x86 Flags

In the preceding example, the two-letter codes at the end of the second line are *flags*. These are single-bit registers and have a variety of uses.

The following table lists the x86 flags:

Flag Code Flag Name Value Flag Status Status Description **of**

Overflow Flag

0 1 **nvov**

No overflow Overflow **df**

Direction Flag

0 1 **updn**

Direction up Direction down **if**

Interrupt Flag

0 1 **diei**

Interrupts disabled Interrupts enabled **sf**

Sign Flag

0 1 **plng**

Positive (or zero) Negative **zf**

Zero Flag

0 1 **nzzr**

Nonzero Zero **af**

Auxiliary Carry Flag

0 1 **naac**

No auxiliary carry Auxiliary carry **pf**

Parity Flag

0 1 **pepo**

Parity even Parity odd **cf**

Carry Flag

0 1 **nccy**

No carry Carry **tf**

Trap Flag

If **tf** equals 1, the processor will raise a STATUS_SINGLE_STEP exception after the execution of one instruction. This flag is used by a debugger to implement single-step tracing. It should not be used by other applications.

iopl

I/O Privilege Level

This is a two-bit integer, with values between zero and 3. It is used by the operating system to control access to hardware. It should not be used by applications.

When registers are displayed as a result of some command in the Debugger Command window, it is the *flag status* that is displayed. However, if you want to change a flag using the **r (Registers)** command, you should refer to it by the *flag code*.

In the Registers window of WinDbg, the flag code is used to view or alter flags. The flag status is not supported.

Here is an example. In the preceding register display, the flag status **ng** appears. This means that the sign flag is currently set to 1. To change this, use the following command:

```
r sf=0
```

This sets the sign flag to zero. If you do another register display, the **ng** status code will not appear. Instead, the **pl** status code will be displayed.

The Sign Flag, Zero Flag, and Carry Flag are the most commonly-used flags.

Conditions

A *condition* describes the state of one or more flags. All conditional operations on the x86 are expressed in terms of conditions.

The assembler uses a one or two letter abbreviation to represent a condition. A condition can be represented by multiple abbreviations. For example, AE ("above or equal") is the same condition as NB ("not below"). The following table lists some common conditions and their meaning.

CONDITION NAME	FLAGS	MEANING
Z	ZF=1	Result of last operation was zero.
NZ	ZF=0	Result of last operation was not zero.
C	CF=1	Last operation required a carry or borrow. (For unsigned integers, this indicates overflow.)
NC	CF=0	Last operation did not require a carry or borrow. (For unsigned integers, this indicates overflow.)
S	SF=1	Result of last operation has its high bit set.
NS	SF=0	Result of last operation has its high bit clear.
O	OF=1	When treated as a signed integer operation, the last operation caused an overflow or underflow.
NO	OF=0	When treated as signed integer operation, the last operation did not cause an overflow or underflow.

Conditions can also be used to compare two values. The **cmp** instruction compares its two operands, and then sets flags as if subtracted one operand from the other. The following conditions can be used to check the result of **cmp** *value1*, *value2*.

CONDITION NAME	FLAGS	MEANING AFTER A CMP OPERATION.
E	ZF=1	*value1* == *value2*.

CONDITION NAME	FLAGS	MEANING AFTER A CMP OPERATION.
NE	ZF=0	*value1* != *value2*.
GE NL	SF=OF	*value1* >= *value2*. Values are treated as signed integers.
LE NG	ZF=1 or SF!=OF	*value1* <= *value2*. Values are treated as signed integers.
G NLE	ZF=0 and SF=OF	*value1* > *value2*. Values are treated as signed integers.
L NGE	SF!=OF	*value1* < *value2*. Values are treated as signed integers.
AE NB	CF=0	*value1* >= *value2*. Values are treated as unsigned integers.
BE NA	CF=1 or ZF=1	*value1* <= *value2*. Values are treated as unsigned integers.
A NBE	CF=0 and ZF=0	*value1* > *value2*. Values are treated as unsigned integers.
B NAE	CF=1	*value1* < *value2*. Values are treated as unsigned integers.

Conditions are typically used to act on the result of a **cmp** or **test** instruction. For example,

```
cmp eax, 5
jz equal
```

compares the **eax** register against the number 5 by computing the expression (**eax** - 5) and setting flags according to the result. If the result of the subtraction is zero, then the **zr** flag will be set, and the **jz** condition will be true so the jump will be taken.

Data Types

- byte: 8 bits
- word: 16 bits
- dword: 32 bits
- qword: 64 bits (includes floating-point doubles)
- tword: 80 bits (includes floating-point extended doubles)
- oword: 128 bits

Notation

The following table indicates the notation used to describe assembly language instructions.

NOTATION	MEANING
r, **r1**, **r2**...	Registers
m	Memory address (see the succeeding Addressing Modes section for more information.)
#n	Immediate constant
r/m	Register or memory
r/#n	Register or immediate constant
r/m/#n	Register, memory, or immediate constant
cc	A condition code listed in the preceding Conditions section.
T	"B", "W", or "D" (byte, word or dword)
acc*T*	Size *T* accumulator: **al** if *T* = "B", **ax** if *T* = "W", or **eax** if *T* = "D"

Addressing Modes

There are several different addressing modes, but they all take the form **T ptr [expr]**, where **T** is some data type (see the preceding Data Types section) and **expr** is some expression involving constants and registers.

The notation for most modes can be deduced without much difficulty. For example, **BYTE PTR [esi+edx*8+3]** means "take the value of the **esi** register, add to it eight times the value of the **edx** register, add three, then access the byte at the resulting address."

Pipelining

The Pentium is dual-issue, which means that it can perform up to two actions in one clock tick. However, the rules on when it is capable of doing two actions at once (known as *pairing*) are very complicated.

Because x86 is a CISC processor, you do not have to worry about jump delay slots.

Synchronized Memory Access

Load, modify, and store instructions can receive a **lock** prefix, which modifies the instruction as follows:

1. Before issuing the instruction, the CPU will flush all pending memory operations to ensure coherency. All data prefetches are abandoned.
2. While issuing the instruction, the CPU will have exclusive access to the bus. This ensures the atomicity of the load/modify/store operation.

The **xchg** instruction automatically obeys the previous rules whenever it exchanges a value with memory.

All other instructions default to nonlocking.

Jump Prediction

Unconditional jumps are predicted to be taken.

Conditional jumps are predicted to be taken or not taken, depending on whether they were taken the last time they were executed. The cache for recording jump history is limited in size.

If the CPU does not have a record of whether the conditional jump was taken or not taken the last time it was executed, it predicts backward conditional jumps as taken and forward conditional jumps as not taken.

Alignment

The x86 processor will automatically correct unaligned memory access, at a performance penalty. No exception is raised.

A memory access is considered aligned if the address is an integer multiple of the object size. For example, all BYTE accesses are aligned (everything is an integer multiple of 1), WORD accesses to even addresses are aligned, and DWORD addresses must be a multiple of 4 in order to be aligned.

The **lock** prefix should not be used for unaligned memory accesses.

x86 Instructions

9/25/2019 • 9 minutes to read • Edit Online

In the lists in this section, instructions marked with an asterisk (*) are particularly important. Instructions not so marked are not critical.

On the x86 processor, instructions are variable-sized, so disassembling backward is an exercise in pattern matching. To disassemble backward from an address, you should start disassembling at a point further back than you really want to go, then look forward until the instructions start making sense. The first few instructions may not make any sense because you may have started disassembling in the middle of an instruction. There is a possibility, unfortunately, that the disassembly will never synchronize with the instruction stream and you will have to try disassembling at a different starting point until you find a starting point that works.

For well-packed **switch** statements, the compiler emits data directly into the code stream, so disassembling through a **switch** statement will usually stumble across instructions that make no sense (because they are really data). Find the end of the data and continue disassembling there.

Instruction Notation

The general notation for instructions is to put the destination register on the left and the source on the right. However, there can be some exceptions to this rule.

Arithmetic instructions are typically two-register with the source and destination registers combining. The result is stored into the destination.

Some of the instructions have both 16-bit and 32-bit versions, but only the 32-bit versions are listed here. Not listed here are floating-point instructions, privileged instructions, and instructions that are used only in segmented models (which Microsoft Win32 does not use).

To save space, many of the instructions are expressed in combined form, as shown in the following example.

*	MOV	**r1**, **r**/m/#n	**r1** = **r**/m/#n

means that the first parameter must be a register, but the second can be a register, a memory reference, or an immediate value.

To save even more space, instructions can also be expressed as shown in the following.

*	MOV	**r1**/m, **r**/m/#n	**r1**/m = **r**/m/#n

which means that the first parameter can be a register or a memory reference, and the second can be a register, memory reference, or immediate value.

Unless otherwise noted, when this abbreviation is used, you cannot choose memory for both source and destination.

Furthermore, a bit-size suffix (8, 16, 32) can be appended to the source or destination to indicate that the parameter must be of that size. For example, r8 means an 8-bit register.

Memory, Data Transfer, and Data Conversion

Memory and data transfer instructions do not affect flags.

Effective Address

*	LEA	**r**, m	Load effective address. (r = address of m)

For example, **LEA eax, [esi+4]** means **eax** = **esi** + 4. This instruction is often used to perform arithmetic.

Data Transfer

	MOV	**r1**/m, **r2**/m/#n	**r1**/m = **r**/m/#n
	MOVSX	**r1**, **r**/m	Move with sign extension.
*	MOVZX	**r1**, **r**/m	Move with zero extension.

MOVSX and **MOVZX** are special versions of the **mov** instruction that perform sign extension or zero extension from the source to the destination. This is the only instruction that allows the source and destination to be different sizes. (And in fact, they must be different sizes.

Stack Manipulation

The stack is pointed to by the **esp** register. The value at **esp** is the top of the stack (most recently pushed, first to be popped); older stack elements reside at higher addresses.

	PUSH	**r**/m/#n	Push value onto stack.
	POP	**r**/m	Pop value from stack.
	PUSHFD		Push flags onto stack.
	POPFD		Pop flags from stack.
	PUSHAD		Push all integer registers.
	POPAD		Pop all integer registers.
	ENTER	#n, #n	Build stack frame.
*	LEAVE		Tear down stack frame

The C/C++ compiler does not use the **enter** instruction. (The **enter** instruction is used to implement nested procedures in languages like Algol or Pascal.)

The **leave** instruction is equivalent to:

```
mov esp, ebp
pop ebp
```

Data Conversion

CBW	Convert byte (**al**) to word (**ax**).
CWD	Convert word (**ax**) to dword (**dx**:**ax**).
CWDE	Convert word (**ax**) to dword (**eax**).
CDQ	convert dword (**eax**) to qword (**edx**:**eax**).

All conversions perform sign extension.

Arithmetic and Bit Manipulation

All arithmetic and bit manipulation instructions modify flags.

Arithmetic

	ADD	**r1**/m, **r2**/m/#n	**r1**/m += **r2**/m/#n
	ADC	**r1**/m, **r2**/m/#n	**r1**/m += **r2**/m/#n + carry
	SUB	**r1**/m, **r2**/m/#n	**r1**/m -= **r2**/m/#n
	SBB	**r1**/m, **r2**/m/#n	**r1**/m -= **r2**/m/#n + carry
	NEG	**r1**/m	**r1**/m = -**r1**/m
	INC	**r**/m	**r**/m += 1
	DEC	**r**/m	**r**/m -= 1
	CMP	**r1**/m, **r2**/m/#n	Compute **r1**/m - **r2**/m/#n

The **cmp** instruction computes the subtraction and sets flags according to the result, but throws the result away. It is typically followed by a conditional **jump** instruction that tests the result of the subtraction.

	MUL	**r**/m8	**ax** = **al** * **r**/m8

	MUL	**r**/m16	**dx**:**ax** = **ax** * **r**/m16
	MUL	**r**/m32	**edx**:**eax** = **eax** * **r**/m32
	IMUL	**r**/m8	**ax** = **al** * **r**/m8
	IMUL	**r**/m16	**dx**:**ax** = **ax** * **r**/m16
	IMUL	**r**/m32	**edx**:**eax** = **eax** * **r**/m32
	IMUL	**r1**, **r2**/m	**r1** *= **r2**/m
	IMUL	**r1**, **r2**/m, #n	**r1** = **r2**/m * #n

Unsigned and signed multiplication. The state of flags after multiplication is undefined.

	DIV	**r**/m8	(**ah**, **al**) = (**ax** % **r**/m8, **ax** / **r**/m8)
	DIV	**r**/m16	(**dx**, **ax**) = **dx**:**ax** / **r**/m16
	DIV	**r**/m32	(**edx**, **eax**) = **edx**:**eax** / **r**/m32
	IDIV	**r**/m8	(**ah**, **al**) = **ax** / **r**/m8
	IDIV	**r**/m16	(**dx**, **ax**) = **dx**:**ax** / **r**/m16
	IDIV	**r**/m32	(**edx**, **eax**) = **edx**:**eax** / **r**/m32

Unsigned and signed division. The first register in the pseudocode explanation receives the remainder and the second receives the quotient. If the result overflows the destination, a division overflow exception is generated.

The state of flags after division is undefined.

*	SET*cc*	**r**/m8	Set **r**/m8 to 0 or 1

If the condition *cc* is true, then the 8-bit value is set to 1. Otherwise, the 8-bit value is set to zero.

Binary-coded Decimal

You will not see these instructions unless you are debugging code written in COBOL.

	DAA	Decimal adjust after addition.
	DAS	Decimal adjust after subtraction.

These instructions adjust the **al** register after performing a packed binary-coded decimal operation.

AAA	ASCII adjust after addition.
AAS	ASCII adjust after subtraction.

These instructions adjust the **al** register after performing an unpacked binary-coded decimal operation.

AAM	ASCII adjust after multiplication.
AAD	ASCII adjust after division.

These instructions adjust the **al** and **ah** registers after performing an unpacked binary-coded decimal operation.

Bits

	AND	**r1**/m, **r2**/m/#n	**r1**/m = **r1**/m and **r2**/m/#n
	OR	**r1**/m, **r2**/m/#n	**r1**/m = **r1**/m or **r2**/m/#n
	XOR	**r1**/m, **r2**/m/#n	**r1**/m = **r1**/m xor **r2**/m/#n
	NOT	**r1**/m	**r1**/m = bitwise not **r1**/m
*	TEST	**r1**/m, **r2**/m/#n	Compute **r1**/m and **r2**/m/#n

The **test** instruction computes the logical AND operator and sets flags according to the result, but throws the result away. It is typically followed by a conditional jump instruction that tests the result of the logical AND.

	SHL	**r1**/m, **cl**/#n	**r1**/m <<= **cl**/#n
	SHR	**r1**/m, **cl**/#n	**r1**/m >>= **cl**/#n zero-fill
*	SAR	**r1**/m, **cl**/#n	**r1**/m >>= **cl**/#n sign-fill

The last bit shifted out is placed in the carry.

	SHLD	**r1**, **r2**/m, **cl**/#n	Shift left double.

Shift **r1** left by **cl**/#n, filling with the top bits of **r2**/m. The last bit shifted out is placed in the carry.

SHRD	**r1**, **r2**/m, **cl**/#n	Shift right double.

Shift **r1** right by **cl**/#n, filling with the bottom bits of **r2**/m. The last bit shifted out is placed in the carry.

ROL	**r1**, **cl**/#n	Rotate **r1** left by **cl**/#n.
ROR	**r1**, **cl**/#n	Rotate **r1** right by **cl**/#n.
RCL	**r1**, **cl**/#n	Rotate **r1**/C left by **cl**/#n.
RCR	**r1**, **cl**/#n	Rotate **r1**/C right by **cl**/#n.

Rotation is like shifting, except that the bits that are shifted out reappear as the incoming fill bits. The C-language version of the rotation instructions incorporate the carry bit into the rotation.

BT	**r1**, **r2**/#n	Copy bit **r2**/#n of **r1** into carry.
BTS	**r1**, **r2**/#n	Set bit **r2**/#n of **r1**, copy previous value into carry.
BTC	**r1**, **r2**/#n	Clear bit **r2**/#n of **r1**, copy previous value into carry.

Control Flow

	Jcc	dest	Branch conditional.
	JMP	dest	Jump direct.
	JMP	**r**/m	Jump indirect.
	CALL	dest	Call direct.
*	CALL	**r**/m	Call indirect.

The **call** instruction pushes the return address onto the stack then jumps to the destination.

*	RET	*#n*	Return

The **ret** instruction pops and jumps to the return address on the stack. A nonzero *#n* in the **RET** instruction indicates that after popping the return address, the value *#n* should be added to the stack pointer.

LOOP	Decrement **ecx** and jump if result is nonzero.
LOOPZ	Decrement **ecx** and jump if result is nonzero and **zr** was set.
LOOPNZ	Decrement **ecx** and jump if result is nonzero and **zr** was clear.
JECXZ	Jump if **ecx** is zero.

These instructions are remnants of the x86's CISC heritage and in recent processors are actually slower than the equivalent instructions written out the long way.

String Manipulation

	MOVS*T*	Move *T* from **esi** to **edi.**
	CMPS*T*	Compare *T* from **esi** with **edi.**
	SCAS*T*	Scan *T* from **edi** for acc*T.*
	LODS*T*	Load *T* from **esi** into acc*T.*
	STOS*T*	Store *T* to **edi** from acc*T.*

After performing the operation, the source and destination register are incremented or decremented by sizeof(*T*), according to the setting of the direction flag (up or down).

The instruction can be prefixed by **REP** to repeat the operation the number of times specified by the **ecx** register.

The **rep mov** instruction is used to copy blocks of memory.

The **rep stos** instruction is used to fill a block of memory with acc*T*.

Flags

LAHF	Load **ah** from flags.
SAHF	Store **ah** to flags.
STC	Set carry.
CLC	Clear carry.

CMC	Complement carry.
STD	Set direction to *down*.
CLD	Set direction to *up*.
STI	Enable interrupts.
CLI	Disable interrupts.

Interlocked Instructions

XCHG	**r1**, **r**/m	Swap **r1** and **r**/m.
XADD	**r1**, **r**/m	Add **r1** to **r**/m, put original value in **r1.**
CMPXCHG	**r1**, **r**/m	Compare and exchange conditional.

The **cmpxchg** instruction is the atomic version of the following:

```
   cmp     accT, r/m
   jz      match
   mov     accT, r/m
   jmp     done
match:
   mov     r/m, r1
done:
```

Miscellaneous

	INT	#n	Trap to kernel.
	BOUND	**r**, m	Trap if **r** not in range.
*	NOP		No operation.
	XLATB		**al** = [**ebx** + **al**]
	BSWAP	**r**	Swap byte order in register.

Here is a special case of the **int** instruction.

INT	3	Debugger breakpoint trap.

The opcode for **INT 3** is 0xCC. The opcode for **NOP** is 0x90.

When debugging code, you may need to patch out some code. You can do this by replacing the offending bytes with 0x90.

Idioms

	XOR	**r**, **r**	**r** = 0
	TEST	**r**, **r**	Check if **r** = 0.
*	ADD	**r**, **r**	Shift **r** left by 1.

Annotated x86 Disassembly

12/5/2018 • 13 minutes to read • Edit Online

The following section will walk you through a disassembly example.

Source Code

The following is the code for the function that will be analyzed.

```
HRESULT CUserView::CloseView(void)
{
    if (m_fDestroyed) return S_OK;

    BOOL fViewObjectChanged = FALSE;
    ReleaseAndNull(&m_pdtgt);

    if (m_psv) {
        m_psb->EnableModelessSB(FALSE);
        if(m_pws) m_pws->ViewReleased();

        IShellView* psv;

        HWND hwndCapture = GetCapture();
        if (hwndCapture && hwndCapture == m_hwnd) {
            SendMessage(m_hwnd, WM_CANCELMODE, 0, 0);
        }

        m_fHandsOff = TRUE;
        m_fRecursing = TRUE;
        NotifyClients(m_psv, NOTIFY_CLOSING);
        m_fRecursing = FALSE;

        m_psv->UIActivate(SVUIA_DEACTIVATE);

        psv = m_psv;
        m_psv = NULL;

        ReleaseAndNull(&_pctView);

        if (m_pvo) {
            IAdviseSink *pSink;
            if (SUCCEEDED(m_pvo->GetAdvise(NULL, NULL, &pSink)) && pSink) {
                if (pSink == (IAdviseSink *)this)
                    m_pvo->SetAdvise(0, 0, NULL);
                pSink->Release();
            }

            fViewObjectChanged = TRUE;
            ReleaseAndNull(&m_pvo);
        }

        if (psv) {
            psv->SaveViewState();
            psv->DestroyViewWindow();
            psv->Release();
        }

        m_hwndView = NULL;
        m_fHandsOff = FALSE;

        if (m_pcache) {
            GlobalFree(m_pcache);
            m_pcache = NULL;
```

```
            m_pCache = NULL;
        }

        m_psb->EnableModelessSB(TRUE);

        CancelPendingActions();
    }

    ReleaseAndNull(&_psf);

    if (fViewObjectChanged)
        NotifyViewClients(DVASPECT_CONTENT, -1);

    if (m_pszTitle) {
        LocalFree(m_pszTitle);
        m_pszTitle = NULL;
    }

    SetRect(&m_rcBounds, 0, 0, 0, 0);
    return S_OK;
}
```

Assembly Code

This section contains the annotated disassembly example.

Functions which use the **ebp** register as a frame pointer start out as follows:

```
HRESULT CUserView::CloseView(void)
SAMPLE!CUserView__CloseView:
71517134 55              push    ebp
71517135 8bec            mov     ebp,esp
```

This sets up the frame so the function can access its parameters as positive offsets from **ebp**, and local variables as negative offsets.

This is a method on a private COM interface, so the calling convention is __**stdcall**. This means that parameters are pushed right to left (in this case, there are none), the "this" pointer is pushed, and then the function is called. Thus, upon entry into the function, the stack looks like this:

```
[esp+0] = return address
[esp+4] = this
```

After the two preceding instructions, the parameters are accessible as:

```
[ebp+0] = previous ebp pushed on stack
[ebp+4] = return address
[ebp+8] = this
```

For a function that uses **ebp** as a frame pointer, the first pushed parameter is accessible at [**ebp**+8]; subsequent parameters are accessible at consecutive higher DWORD addresses.

```
71517137 51              push    ecx
71517138 51              push    ecx
```

This function requires only two local stack variables, so a **sub esp**, 8 instruction. The pushed values are then available as [**ebp**-4] and [**ebp**-8].

For a function that uses **ebp** as a frame pointer, stack local variables are accessible at negative offsets from the

ebp register.

```
71517139 56              push    esi
```

Now the compiler saves the registers that are required to be preserved across function calls. Actually, it saves them in bits and pieces, interleaved with the first line of actual code.

```
7151713a 8b7508          mov     esi,[ebp+0x8]     ; esi = this
7151713d 57              push    edi               ; save another registers
```

It so happens that CloseView is a method on ViewState, which is at offset 12 in the underlying object. Consequently, **this** is a pointer to a ViewState class, although when there is possible confusion with another base class, it will be more carefully specified as (ViewState*)**this**.

```
    if (m_fDestroyed)
7151713e 33ff            xor     edi,edi           ; edi = 0
```

XORing a register with itself is a standard way of zeroing it out.

```
71517140 39beac000000    cmp     [esi+0xac],edi    ; this->m_fDestroyed == 0?
71517146 7407            jz      NotDestroyed (7151714f)  ; jump if equal
```

The **cmp** instruction compares two values (by subtracting them). The **jz** instruction checks if the result is zero, indicating that the two compared values are equal.

The cmp instruction compares two values; a subsequent j instruction jumps based on the result of the comparison.

```
    return S_OK;
71517148 33c0            xor     eax,eax           ; eax = 0 = S_OK
7151714a e972010000      jmp     ReturnNoEBX (715172c1) ; return, do not pop EBX
```

The compiler delayed saving the EBX register until later in the function, so if the program is going to "early-out" on this test, then the exit path needs to be the one that does not restore EBX.

```
    BOOL fViewObjectChanged = FALSE;
    ReleaseAndNull(&m_pdtgt);
```

The execution of these two lines of code is interleaved, so pay attention.

```
NotDestroyed:
7151714f 8d86c0000000    lea     eax,[esi+0xc0]    ; eax = &m_pdtgt
```

The **lea** instruction computes the effect address of a memory access and stores it in the destination. The actual memory address is not dereferenced.

The lea instruction takes the address of a variable.

```
71517155 53              push    ebx
```

You should save that EBX register before it is damaged.

```
71517156 8b1d10195071     mov ebx,[_imp__ReleaseAndNull]
```

Because you will be calling **ReleaseAndNull** frequently, it is a good idea to cache its address in EBX.

```
7151715c 50               push    eax               ; parameter to ReleaseAndNull
7151715d 897dfc           mov     [ebp-0x4],edi     ; fViewObjectChanged = FALSE
71517160 ffd3             call    ebx               ; call ReleaseAndNull
    if (m_psv) {
71517162 397e74           cmp     [esi+0x74],edi    ; this->m_psv == 0?
71517165 0f8411010000     je      No_Psv (7151727c) ; jump if zero
```

Remember that you zeroed out the EDI register a while back and that EDI is a register preserved across function calls (so the call to **ReleaseAndNull** did not change it). Therefore, it still holds the value zero and you can use it to quickly test for zero.

```
        m_psb->EnableModelessSB(FALSE);
7151716b 8b4638           mov     eax,[esi+0x38]    ; eax = this->m_psb
7151716e 57               push    edi               ; FALSE
7151716f 50               push    eax               ; "this" for callee
71517170 8b08             mov     ecx,[eax]         ; ecx = m_psb->lpVtbl
71517172 ff5124           call    [ecx+0x24]        ; __stdcall EnableModelessSB
```

The above pattern is a telltale sign of a COM method call.

COM method calls are pretty popular, so it is a good idea to learn to recognize them. In particular, you should be able to recognize the three IUnknown methods directly from their Vtable offsets: QueryInterface=0, AddRef=4, and Release=8.

```
        if(m_pws) m_pws->ViewReleased();
71517175 8b8614010000     mov     eax,[esi+0x114]   ; eax = this->m_pws
7151717b 3bc7             cmp     eax,edi           ; eax == 0?
7151717d 7406             jz      NoWS (71517185) ; if so, then jump
7151717f 8b08             mov     ecx,[eax]         ; ecx = m_pws->lpVtbl
71517181 50               push    eax               ; "this" for callee
71517182 ff510c           call    [ecx+0xc]         ; __stdcall ViewReleased
NoWS:
        HWND hwndCapture = GetCapture();
71517185 ff15e01a5071    call [_imp__GetCapture]    ; call GetCapture
```

Indirect calls through globals is how function imports are implemented in Microsoft Win32. The loader fixes up the globals to point to the actual address of the target. This is a handy way to get your bearings when you are investigating a crashed machine. Look for the calls to imported functions and in the target. You will usually have the name of some imported function, which you can use to determine where you are in the source code.

```
        if (hwndCapture && hwndCapture == m_hwnd) {
            SendMessage(m_hwnd, WM_CANCELMODE, 0, 0);
        }
7151718b 3bc7             cmp     eax,edi           ; hwndCapture == 0?
7151718d 7412             jz      No_Capture (715171a1) ; jump if zero
```

The function return value is placed in the EAX register.

```
7151718f 8b4e44          mov     ecx,[esi+0x44]    ; ecx = this->m_hwnd
71517192 3bc1            cmp     eax,ecx           ; hwndCapture = ecx?
71517194 750b            jnz     No_Capture (715171a1) ; jump if not

71517196 57              push    edi               ; 0
71517197 57              push    edi               ; 0
71517198 6a1f            push    0x1f              ; WM_CANCELMODE
7151719a 51              push    ecx               ; hwndCapture
7151719b ff1518195071    call    [_imp__SendMessageW] ; SendMessage
No_Capture:
        m_fHandsOff = TRUE;
        m_fRecursing = TRUE;
715171a1 66818e0c0100000180 or    word ptr [esi+0x10c],0x8001 ; set both flags at once

        NotifyClients(m_psv, NOTIFY_CLOSING);
715171aa 8b4e20          mov     ecx,[esi+0x20]    ; ecx = (CNotifySource*)this.vtbl
715171ad 6a04            push    0x4               ; NOTIFY_CLOSING
715171af 8d4620          lea     eax,[esi+0x20]    ; eax = (CNotifySource*)this
715171b2 ff7674          push    [esi+0x74]        ; m_psv
715171b5 50              push    eax               ; "this" for callee
715171b6 ff510c          call    [ecx+0xc]         ; __stdcall NotifyClients
```

Notice how you had to change your "this" pointer when calling a method on a different base class from your own.

```
        m_fRecursing = FALSE;
715171b9 80a60d0100007f   and     byte ptr [esi+0x10d],0x7f
        m_psv->UIActivate(SVUIA_DEACTIVATE);
715171c0 8b4674          mov     eax,[esi+0x74]    ; eax = m_psv
715171c3 57              push    edi               ; SVUIA_DEACTIVATE = 0
715171c4 50              push    eax               ; "this" for callee
715171c5 8b08            mov     ecx,[eax]         ; ecx = vtbl
715171c7 ff511c          call    [ecx+0x1c]        ; __stdcall UIActivate
        psv = m_psv;
        m_psv = NULL;
715171ca 8b4674          mov     eax,[esi+0x74]    ; eax = m_psv
715171cd 897e74          mov     [esi+0x74],edi    ; m_psv = NULL
715171d0 8945f8          mov     [ebp-0x8],eax     ; psv = eax
```

The first local variable is **psv**.

```
        ReleaseAndNull(&_pctView);
715171d3 8d466c          lea     eax,[esi+0x6c]    ; eax = &_pctView
715171d6 50              push    eax               ; parameter
715171d7 ffd3            call    ebx               ; call ReleaseAndNull
        if (m_pvo) {
715171d9 8b86a8000000    mov     eax,[esi+0xa8]    ; eax = m_pvo
715171df 8dbea8000000    lea     edi,[esi+0xa8]    ; edi = &m_pvo
715171e5 85c0            test    eax,eax           ; eax == 0?
715171e7 7448            jz      No_Pvo (71517231) ; jump if zero
```

Note that the compiler speculatively prepared the address of the **m_pvo** member, because you are going to use it frequently for a while. Thus, having the address handy will result in smaller code.

```
           if (SUCCEEDED(m_pvo->GetAdvise(NULL, NULL, &pSink)) && pSink) {
715171e9 8b08            mov     ecx,[eax]         ; ecx = m_pvo->lpVtbl
715171eb 8d5508          lea     edx,[ebp+0x8]     ; edx = &pSink
715171ee 52              push    edx               ; parameter
715171ef 6a00            push    0x0               ; NULL
715171f1 6a00            push    0x0               ; NULL
715171f3 50              push    eax               ; "this" for callee
715171f4 ff5120          call    [ecx+0x20]        ; __stdcall GetAdvise
715171f7 85c0            test    eax,eax           ; test bits of eax
715171f9 7c2c            jl      No_Advise (71517227) ; jump if less than zero
715171fb 33c9            xor     ecx,ecx           ; ecx = 0
715171fd 394d08          cmp     [ebp+0x8],ecx     ; _pSink == ecx?
71517200 7425            jz      No_Advise (71517227)
```

Notice that the compiler concluded that the incoming "this" parameter was not required (because it long ago stashed that into the ESI register). Thus, it reused the memory as the local variable pSink.

If the function uses an EBP frame, then incoming parameters arrive at positive offsets from EBP and local variables are placed at negative offsets. But, as in this case, the compiler is free to reuse that memory for any purpose.

If you are paying close attention, you will see that the compiler could have optimized this code a little better. It could have delayed the **lea edi, [esi+0xa8]** instruction until after the two **push 0x0** instructions, replacing them with **push edi**. This would have saved 2 bytes.

```
               if (pSink == (IAdviseSink *)this)
```

These next several lines are to compensate for the fact that in C++, (IAdviseSink *)**NULL** must still be **NULL**. So if your "this" is really "(ViewState*)NULL", then the result of the cast should be **NULL** and not the distance between IAdviseSink and IBrowserService.

```
71517202 8d46ec          lea     eax,[esi-0x14]    ; eax = -(IAdviseSink*)this
71517205 8d5614          lea     edx,[esi+0x14]    ; edx = (IAdviseSink*)this
71517208 f7d8            neg     eax               ; eax = -eax (sets carry if != 0)
7151720a 1bc0            sbb     eax,eax           ; eax = eax - eax - carry
7151720c 23c2            and     eax,edx           ; eax = NULL or edx
```

Although the Pentium has a conditional move instruction, the base i386 architecture does not, so the compiler uses specific techniques to simulate a conditional move instruction without taking any jumps.

The general pattern for a conditional evaluation is the following:

```
        neg     r
        sbb     r, r
        and     r, (val1 - val2)
        add     r, val2
```

The **neg r** sets the carry flag if **r** is nonzero, because **neg** negates the value by subtracting from zero. And, subtracting from zero will generate a borrow (set the carry) if you subtract a nonzero value. It also damages the value in the **r** register, but that is acceptable because you are about to overwrite it anyway.

Next, the **sbb r, r** instruction subtracts a value from itself, which always results in zero. However, it also subtracts the carry (borrow) bit, so the net result is to set **r** to zero or -1, depending on whether the carry was clear or set, respectively.

Therefore, **sbb r, r** sets **r** to zero if the original value of **r** was zero, or to -1 if the original value was nonzero.

The third instruction performs a mask. Because the **r** register is zero or -1, "this" serves either to leave **r** zero or to change **r** from -1 to **(val1 - val1)**, in that ANDing any value with -1 leaves the original value.

Therefore, the result of "and r, (val1 - val1)" is to set r to zero if the original value of r was zero, or to "(val1 - val2)" if the original value of r was nonzero.

Finally, you add **val2** to **r**, resulting in **val2** or **(val1 - val2) + val2 = val1**.

Thus, the ultimate result of this series of instructions is to set **r** to **val2** if it was originally zero or to **val1** if it was nonzero. This is the assembly equivalent of **r = r ? val1 : val2**.

In this particular instance, you can see that **val2 = 0** and **val1 = (IAdviseSink*)this**. (Notice that the compiler elided the final **add eax, 0** instruction because it has no effect.)

```
7151720e 394508          cmp     [ebp+0x8],eax ; pSink == (IAdviseSink*)this?
71517211 750b            jnz     No_SetAdvise (7151721e) ; jump if not equal
```

Earlier in this section, you set EDI to the address of the **m_pvo** member. You are going to be using it now. You also zeroed out the ECX register earlier.

```
                  m_pvo->SetAdvise(0, 0, NULL);
71517213 8b07            mov     eax,[edi]         ; eax = m_pvo
71517215 51              push    ecx               ; NULL
71517216 51              push    ecx               ; 0
71517217 51              push    ecx               ; 0
71517218 8b10            mov     edx,[eax]         ; edx = m_pvo->lpVtbl
7151721a 50              push    eax               ; "this" for callee
7151721b ff521c          call    [edx+0x1c]        ; __stdcall SetAdvise
No_SetAdvise:
               pSink->Release();
7151721e 8b4508          mov     eax,[ebp+0x8]     ; eax = pSink
71517221 50              push    eax               ; "this" for callee
71517222 8b08            mov     ecx,[eax]         ; ecx = pSink->lpVtbl
71517224 ff5108          call    [ecx+0x8]         ; __stdcall Release
No_Advise:
```

All these COM method calls should look very familiar.

The evaluation of the next two statements is interleaved. Do not forget that EBX contains the address of **ReleaseAndNull**.

```
           fViewObjectChanged = TRUE;
           ReleaseAndNull(&m_pvo);
71517227 57              push    edi               ; &m_pvo
71517228 c745fc01000000  mov     dword ptr [ebp-0x4],0x1 ; fViewObjectChanged = TRUE
7151722f ffd3            call    ebx               ; call ReleaseAndNull
No_Pvo:
        if (psv) {
71517231 8b7df8          mov     edi,[ebp-0x8]     ; edi = psv
71517234 85ff            test    edi,edi           ; edi == 0?
71517236 7412            jz      No_Psv2 (7151724a) ; jump if zero
           psv->SaveViewState();
71517238 8b07            mov     eax,[edi]         ; eax = psv->lpVtbl
7151723a 57              push    edi               ; "this" for callee
7151723b ff5034          call    [eax+0x34]        ; __stdcall SaveViewState
```

Here are more COM method calls.

```
            psv->DestroyViewWindow();
7151723e 8b07             mov     eax,[edi]         ; eax = psv->lpVtbl
71517240 57               push    edi               ; "this" for callee
71517241 ff5028           call    [eax+0x28]        ; __stdcall DestroyViewWindow
            psv->Release();
71517244 8b07             mov     eax,[edi]         ; eax = psv->lpVtbl
71517246 57               push    edi               ; "this" for callee
71517247 ff5008           call    [eax+0x8]         ; __stdcall Release
No_Psv2:
        m_hwndView = NULL;
7151724a 83667c00         and     dword ptr [esi+0x7c],0x0 ; m_hwndView = 0
```

ANDing a memory location with zero is the same as setting it to zero, because anything AND zero is zero. The compiler uses this form because, even though it is slower, it is much shorter than the equivalent **mov** instruction. (This code was optimized for size, not speed.)

```
        m_fHandsOff = FALSE;
7151724e 83a60c010000fe   and     dword ptr [esi+0x10c],0xfe
        if (m_pcache) {
71517255 8b4670           mov     eax,[esi+0x70]    ; eax = m_pcache
71517258 85c0             test    eax,eax           ; eax == 0?
7151725a 740b             jz      No_Cache (71517267) ; jump if zero
            GlobalFree(m_pcache);
7151725c 50               push    eax               ; m_pcache
7151725d ff15b4135071     call    [_imp__GlobalFree]    ; call GlobalFree
            m_pcache = NULL;
71517263 83667000         and     dword ptr [esi+0x70],0x0 ; m_pcache = 0
No_Cache:
        m_psb->EnableModelessSB(TRUE);
71517267 8b4638           mov     eax,[esi+0x38]    ; eax = this->m_psb
7151726a 6a01             push    0x1               ; TRUE
7151726c 50               push    eax               ; "this" for callee
7151726d 8b08             mov     ecx,[eax]         ; ecx = m_psb->lpVtbl
7151726f ff5124           call    [ecx+0x24]        ; __stdcall EnableModelessSB
        CancelPendingActions();
```

In order to call **CancelPendingActions**, you have to move from (ViewState*)this to (CUserView*)this. Note also that **CancelPendingActions** uses the __thiscall calling convention instead of __stdcall. According to __thiscall, the "this" pointer is passed in the ECX register instead of being passed on the stack.

```
71517272 8d4eec           lea     ecx,[esi-0x14]    ; ecx = (CUserView*)this
71517275 e832fbffff       call CUserView::CancelPendingActions (71516dac) ; __thiscall
    ReleaseAndNull(&_psf);
7151727a 33ff             xor     edi,edi           ; edi = 0 (for later)
No_Psv:
7151727c 8d4678           lea     eax,[esi+0x78]    ; eax = &_psf
7151727f 50               push    eax               ; parameter
71517280 ffd3             call    ebx               ; call ReleaseAndNull
    if (fViewObjectChanged)
71517282 397dfc           cmp     [ebp-0x4],edi     ; fViewObjectChanged == 0?
71517285 740d             jz      NoNotifyViewClients (71517294) ; jump if zero
       NotifyViewClients(DVASPECT_CONTENT, -1);
71517287 8b46ec           mov     eax,[esi-0x14]    ; eax = ((CUserView*)this)->lpVtbl
7151728a 8d4eec           lea     ecx,[esi-0x14]    ; ecx = (CUserView*)this
7151728d 6aff             push    0xff              ; -1
7151728f 6a01             push    0x1               ; DVASPECT_CONTENT = 1
71517291 ff5024           call    [eax+0x24]        ; __thiscall NotifyViewClients
NoNotifyViewClients:
    if (m_pszTitle)
71517294 8b8680000000     mov     eax,[esi+0x80]    ; eax = m_pszTitle
7151729a 8d9e80000000     lea     ebx,[esi+0x80]    ; ebx = &m_pszTitle (for later)
715172a0 3bc7             cmp     eax,edi           ; eax == 0?
715172a2 7409             jz      No_Title (715172ad) ; jump if zero
        LocalFree(m_pszTitle);
715172a4 50               push    eax               ; m_pszTitle
715172a5 ff1538125071     call   [_imp__LocalFree]
        m_pszTitle = NULL;
```

Remember that EDI is still zero and EBX is still &m_pszTitle, because those registers are preserved by function calls.

```
715172ab 893b             mov     [ebx],edi         ; m_pszTitle = 0
No_Title:
    SetRect(&m_rcBounds, 0, 0, 0, 0);
715172ad 57               push    edi               ; 0
715172ae 57               push    edi               ; 0
715172af 57               push    edi               ; 0
715172b0 81c6fc000000     add     esi,0xfc          ; esi = &this->m_rcBounds
715172b6 57               push    edi               ; 0
715172b7 56               push    esi               ; &m_rcBounds
715172b8 ff15e41a5071     call   [_imp__SetRect]
```

Notice that you do not need the value of "this" any more, so the compiler uses the **add** instruction to modify it in place instead of using up another register to hold the address. This is actually a performance win due to the Pentium u/v pipelining, because the v pipe can do arithmetic, but not address computations.

```
    return S_OK;
715172be 33c0             xor     eax,eax           ; eax = S_OK
```

Finally, you restore the registers you are required to preserve, clean up the stack, and return to your caller, removing the incoming parameters.

```
715172c0 5b               pop     ebx               ; restore
ReturnNoEBX:
715172c1 5f               pop     edi               ; restore
715172c2 5e               pop     esi               ; restore
715172c3 c9               leave                     ; restores EBP and ESP simultaneously
715172c4 c20400           ret     0x4               ; return and clear parameters
```

The x64 Processor

12/5/2018 • 2 minutes to read • Edit Online

This section includes:

x64 Architecture

x64 Instructions

Annotated x64 Disassembly

Note The x64 processor architecture is sometimes referred to as "AMD64", "x86-64", "AMD x86-64" or "Intel64".

x64 Architecture

12/5/2018 • 3 minutes to read • Edit Online

The x64 architecture is a backwards-compatible extension of x86. It provides a legacy 32-bit mode, which is identical to x86, and a new 64-bit mode.

The term "x64" includes both AMD 64 and Intel64. The instruction sets are close to identical.

Registers

x64 extends x86's 8 general-purpose registers to be 64-bit, and adds 8 new 64-bit registers. The 64-bit registers have names beginning with "r", so for example the 64-bit extension of **eax** is called **rax**. The new registers are named **r8** through **r15**.

The lower 32 bits, 16 bits, and 8 bits of each register are directly addressable in operands. This includes registers, like **esi**, whose lower 8 bits were not previously addressable. The following table specifies the assembly-language names for the lower portions of 64-bit registers.

64-BIT REGISTER	LOWER 32 BITS	LOWER 16 BITS	LOWER 8 BITS
rax	**eax**	**ax**	**al**
rbx	**ebx**	**bx**	**bl**
rcx	**ecx**	**cx**	**cl**
rdx	**edx**	**dx**	**dl**
rsi	**esi**	**si**	**sil**
rdi	**edi**	**di**	**dil**
rbp	**ebp**	**bp**	**bpl**
rsp	**esp**	**sp**	**spl**
r8	**r8d**	**r8w**	**r8b**
r9	**r9d**	**r9w**	**r9b**
r10	**r10d**	**r10w**	**r10b**
r11	**r11d**	**r11w**	**r11b**

64-BIT REGISTER	LOWER 32 BITS	LOWER 16 BITS	LOWER 8 BITS
r12	**r12d**	**r12w**	**r12b**
r13	**r13d**	**r13w**	**r13b**
r14	**r14d**	**r14w**	**r14b**
r15	**r15d**	**r15w**	**r15b**

Operations that output to a 32-bit subregister are automatically zero-extended to the entire 64-bit register. Operations that output to 8-bit or 16-bit subregisters are *not* zero-extended (this is compatible x86 behavior).

The high 8 bits of **ax**, **bx**, **cx**, and **dx** are still addressable as **ah**, **bh**, **ch**, **dh**, but cannot be used with all types of operands.

The instruction pointer, **eip**, and **flags** register have been extended to 64 bits (**rip** and **rflags**, respectively) as well.

The x64 processor also provides several sets of floating-point registers:

- Eight 80-bit x87 registers.
- Eight 64-bit MMX registers. (These overlap with the x87 registers.)
- The original set of eight 128-bit SSE registers is increased to sixteen.

Calling Conventions

Unlike the x86, the C/C++ compiler only supports one calling convention on x64. This calling convention takes advantage of the increased number of registers available on x64:

- The first four integer or pointer parameters are passed in the **rcx**, **rdx**, **r8**, and **r9** registers.
- The first four floating-point parameters are passed in the first four SSE registers, **xmm0**-**xmm3**.
- The caller reserves space on the stack for arguments passed in registers. The called function can use this space to spill the contents of registers to the stack.
- Any additional arguments are passed on the stack.
- An integer or pointer return value is returned in the **rax** register, while a floating-point return value is returned in **xmm0**.
- **rax**, **rcx**, **rdx**, **r8**-**r11** are volatile.
- **rbx**, **rbp**, **rdi**, **rsi**, **r12**-**r15** are nonvolatile.

The calling convention for C++ is very similar: the **this** pointer is passed as an implicit first parameter. The next three parameters are passed in registers, while the rest are passed on the stack.

Addressing Modes

The addressing modes in 64-bit mode are similar to, but not identical to, x86.

- Instructions that refer to 64-bit registers are automatically performed with 64-bit precision. (For example **mov rax, [rbx]** moves 8 bytes beginning at **rbx** into **rax**.)
- A special form of the **mov** instruction has been added for 64-bit immediate constants or constant addresses. For all other instructions, immediate constants or constant addresses are still 32 bits.

- x64 provides a new **rip**-relative addressing mode. Instructions that refer to a single constant address are encoded as offsets from **rip**. For example, the **mov rax, [***addr***]** instruction moves 8 bytes beginning at *addr* + **rip** to **rax**.

Instructions, such as **jmp**, **call**, **push**, and **pop**, that implicitly refer to the instruction pointer and the stack pointer treat them as 64 bits registers on x64.

See Also

X86-64 Wikipedia

AMD 64 Developer Resources

x64 Instructions

12/5/2018 • 2 minutes to read • Edit Online

Most x86 instructions continue to be valid for x64 in 64-bit mode. Some rarely-used operations are no longer supported in 64-bit mode, such as:

- binary-coded decimal arithmetic instructions: AAA, AAD, AAM, AAS, DAA, DAS
- BOUND
- PUSHAD and POPAD
- most operations that dealt with segment registers, such as PUSH DS and POP DS. (Operations that use the FS or GS segment registers are still valid.)

The x64 instruction set includes recent additions to the x86, such as SSE 2. Programs compiled for x64 can freely use these instructions.

Data Transfer

The x64 provides new variants of the MOV instruction that can handle 64-bit immediate constants or memory addresses.

MOV	**r**,#n	**r** = #n
MOV	**rax**, m	Move contents at 64-bit address to **rax**.
MOV	m, **rax**	Move contents of **rax** to 64-bit address.

The x64 also provides a new instruction to sign-extend 32-bit operands to 64 bits.

MOVSXD	**r1**, **r**/m	Move DWORD with sign extension to QWORD.

Ordinary MOV operations into 32-bit subregisters automatically zero extend to 64 bits, so there is no MOVZXD instruction.

Two SSE instructions can be used to move 128-bit values (such as GUIDs) from memory to an **xmm***n* register or vice versa.

MOVDQA	**r1**/m, **r2**/m	Move 128-bit aligned value to **xmm***n* register, or vice versa.
MOVDQU	**r1**/m, **r2**/m	Move 128-bit value (not necessarily aligned) to register, or vice versa.

Data Conversion

CDQE	Convert dword (**eax**) to qword (**rax**).
CQO	convert qword (**rax**) to oword (**rdx**:**rax**).

String Manipulation

MOVSQ	Move qword from **rsi** to **rdi.**
CMPSQ	Compare qword at **rsi** with **rdi.**
SCASQ	Scan qword at **rdi**. Compares qword at **rdi** to **rax**.
LODSQ	Load qword from **rsi** into **rax**.
STOSQ	Store qword to **rdi** from **rax**.

Annotated x64 Disassembly

12/5/2018 • 7 minutes to read • Edit Online

The following very simple function illustrates the x64 calling convention.

```
int Simple(int i, int j)
{
    return i*5 + j + 3;
}
```

This compiles to code like this:

```
01001080 lea     eax,[rdx+rcx*4]        ; eax = rdx+rcx*4
01001083 lea     eax,[rcx+rax+0x3]      ; eax = rcx+rax+3
01001087 ret
```

The *i* and *j* parameters are passed in the **ecx** and **edx** registers, respectively. Since there are only two parameters, the routine does not use the stack at all.

The particular code generated exploits three tricks, one of which is specific to the x64:

1. The **lea** operation can be used to perform a series of simple arithmetic operations as a single operation. The first instruction stores $j+i*4$ in **eax**, and the second instruction adds $i+3$ to the result, for a total of $j+i*5+3$.
2. Many operations, such as addition and multiplication, can be done with extra precision, and then truncated to the correct precision. In this instance, the code uses 64-bit addition and multiplication. We can safely truncate the result to 32 bits.
3. On the x64, any operation that outputs to a 32-bit register automatically zero-extends the result. In this case, outputting to **eax** has the effect of truncating the result to 32 bits.

Return values are passed in the **rax** register. In this case, the result is already in the **rax** register, so the function returns.

Next we consider a more complicated function to demonstrate typical x64 disassembly:

```
HRESULT Meaningless(IDispatch *pdisp, DISPID dispid, BOOL fUnique, LPCWSTR pszExe)
{
    IQueryAssociations *pqa;
    HRESULT hr = AssocCreate(CLSID_QueryAssociations, IID_IQueryAssociations, (void**)&pqa);
    if (SUCCEEDED(hr)) {
        hr = pqa->Init(ASSOCF_INIT_BYEXENAME, pszExe, NULL, NULL);
        if (SUCCEEDED(hr)) {
            WCHAR wszName[MAX_PATH];
            DWORD cchName = MAX_PATH;
            hr = pqa->GetString(0, ASSOCSTR_FRIENDLYAPPNAME, NULL, wszName, &cchName);
            if (SUCCEEDED(hr)) {
                VARIANTARG rgvarg[2] = { 0 };
                V_VT(&rgvarg[0]) = VT_BSTR;
                V_BSTR(&rgvarg[0]) = SysAllocString(wszName);
                if (V_BSTR(&rgvarg[0])) {
                    DISPPARAMS dp;
                    LONG lUnique = InterlockedIncrement(&lCounter);
                    V_VT(&rgvarg[1]) = VT_I4;
                    V_I4(&rgvarg[1]) = fUnique ? lUnique : 0;
                    dp.rgvarg = rgvarg;
                    dp.cArgs = 2;
                    dp.rgdispidNamedArgs = NULL;
                    dp.cNamedArgs = 0;
                    hr = pdisp->Invoke(dispid, IID_NULL, 0, DISPATCH_METHOD, &dp, NULL, NULL, NULL);
                    VariantClear(&rgvarg[0]);
                    VariantClear(&rgvarg[1]);
                } else {
                    hr = E_OUTOFMEMORY;
                }
            }
        }
        pqa->Release();
    }
    return hr;
}
```

We'll go through this function and the equivalent assembly line by line.

When entered, the function's parameters are stored as follows:

- **rcx** = *pdisp*.
- **rdx** = *dispid*.
- **r8** = *fUnique*.
- **r9** = *pszExe*.

Recall that the first four parameters are passed in registers. Since this function has only four registers, none are passed on the stack.

The assembly begins as follows:

```
Meaningless:
010010e0 push    rbx                    ; save
010010e1 push    rsi                    ; save
010010e2 push    rdi                    ; save
010010e3 push    r12d                   ; save
010010e5 push    r13d                   ; save
010010e7 push    r14d                   ; save
010010e9 push    r15d                   ; save
010010eb sub     rsp,0x2c0              ; reserve stack
010010f2 mov     rbx,r9                 ; rbx = pszExe
010010f5 mov     r12d,r8d               ; r12 = fUnique (zero-extend)
010010f8 mov     r13d,edx               ; r13 = dispid  (zero-extend)
010010fb mov     rsi,rcx                ; rsi = pdisp
```

The function begins by saving nonvolatile registers, and then reserving stack space for local variables. It then saves parameters in nonvolatile registers. Note that the destination of the middle two **mov** instructions is a 32-bit register, so they are implicitly zero-extended to 64 bits.

```
    IQueryAssociations *pqa;
    HRESULT hr = AssocCreate(CLSID_QueryAssociations, IID_IQueryAssociations, (void**)&pqa);
```

The first parameter to **AssocCreate** is a 128-bit CLSID passed by value. Since this doesn't fit in a 64-bit register, the CLSID is copied to the stack, and a pointer to the stack location is passed instead.

```
010010fe movdqu  xmm0,oword ptr [CLSID_QueryAssociations (01001060)]
01001106 movdqu  oword ptr [rsp+0x60],xmm0  ; temp buffer for first parameter
0100110c lea     r8,[rsp+0x58]          ; arg3 = &pqa
01001111 lea rdx,[IID_IQueryAssociations (01001070)] ; arg2 = &IID_IQueryAssociations
01001118 lea     rcx,[rsp+0x60]         ; arg1 = &temporary
0100111d call qword ptr [_imp_AssocCreate (01001028)] ; call
```

The **movdqu** instruction transfers 128-bits values to and from **xmm***n* registers. In this instance, the assembly code uses it to copy the CLSID to the stack. The pointer to the CLSID is passed in **r8**. The other two arguments are passed in **rcx** and **rdx**.

```
    if (SUCCEEDED(hr)) {

01001123 test    eax,eax
01001125 jl      ReturnEAX (01001281)
```

The code checks to see if the return value is a success.

```
        hr = pqa->Init(ASSOCF_INIT_BYEXENAME, pszExe, NULL, NULL);

0100112b mov     rcx,[rsp+0x58]         ; arg1 = pqa
01001130 mov     rax,[rcx]              ; rax = pqa.vtbl
01001133 xor     r14d,r14d              ; r14 = 0
01001136 mov     [rsp+0x20],r14         ; arg5 = 0
0100113b xor     r9d,r9d                ; arg4 = 0
0100113e mov     r8,rbx                 ; arg3 = pszExe
01001141 mov     r15d,0x2               ; r15 = 2 (for later)
01001147 mov     edx,r15d               ; arg2 = 2 (ASSOCF_INIT_BY_EXENAME)
0100114a call    qword ptr [rax+0x18]   ; call Init method
```

This is an indirect function call using a C++ vtable. The **this** pointer is passed in **rcx** as the first parameter. The first three parameters are passed in registers, while the final parameter is passed on the stack. The function reserves 16 bytes for the parameters passed in registers, so the fifth parameter begins at **rsp**+0x20.

```
        if (SUCCEEDED(hr)) {

0100114d mov     ebx,eax                ; ebx = hr
0100114f test    ebx,ebx                ; FAILED?
01001151 jl      ReleasePQA (01001274)  ; jump if so
```

The assembly-language code saves the result in **ebx**, and checks to see if it's a success code.

```
            WCHAR wszName[MAX_PATH];
            DWORD cchName = MAX_PATH;
            hr = pqa->GetString(0, ASSOCSTR_FRIENDLYAPPNAME, NULL, wszName, &cchName);
            if (SUCCEEDED(hr)) {

01001157 mov     dword ptr [rsp+0x50],0x104 ; cchName = MAX_PATH
0100115f mov     rcx,[rsp+0x58]         ; arg1 = pqa
01001164 mov     rax,[rcx]              ; rax = pqa.vtbl
01001167 lea     rdx,[rsp+0x50]         ; rdx = &cchName
0100116c mov     [rsp+0x28],rdx         ; arg6 = cchName
01001171 lea     rdx,[rsp+0xb0]         ; rdx = &wszName[0]
01001179 mov     [rsp+0x20],rdx         ; arg5 = &wszName[0]
0100117e xor     r9d,r9d                ; arg4 = 0
01001181 mov     r8d,0x4                ; arg3 = 4 (ASSOCSTR_FRIENDLYNAME)
01001187 xor     edx,edx                ; arg2 = 0
01001189 call    qword ptr [rax+0x20]   ; call GetString method
0100118c mov     ebx,eax                ; ebx = hr
0100118e test    ebx,ebx                ; FAILED?
01001190 jl      ReleasePQA (01001274)  ; jump if so
```

Once again, we set up the parameters and call a function, then test the return value for success.

```
                VARIANTARG rgvarg[2] = { 0 };

01001196 lea     rdi,[rsp+0x82]         ; rdi = &rgvarg
0100119e xor     eax,eax                ; rax = 0
010011a0 mov     ecx,0x2e               ; rcx = sizeof(rgvarg)
010011a5 rep     stosb                  ; Zero it out
```

The idiomatic method for zeroing out a buffer on x64 is the same as x86.

```
                V_VT(&rgvarg[0]) = VT_BSTR;
                V_BSTR(&rgvarg[0]) = SysAllocString(wszName);
                if (V_BSTR(&rgvarg[0])) {

010011a7 mov     word ptr [rsp+0x80],0x8 ; V_VT(&rgvarg[0]) = VT_BSTR
010011b1 lea     rcx,[rsp+0xb0]         ; arg1 = &wszName[0]
010011b9 call    qword ptr [_imp_SysAllocString (01001010)] ; call
010011bf mov     [rsp+0x88],rax         ; V_BSTR(&rgvarg[0]) = result
010011c7 test    rax,rax                ; anything allocated?
010011ca je      OutOfMemory (0100126f) ; jump if failed

                   DISPPARAMS dp;
                   LONG lUnique = InterlockedIncrement(&lCounter);

010011d0 lea     rax,[lCounter (01002000)]
010011d7 mov     ecx,0x1
010011dc lock    xadd [rax],ecx             ; interlocked exchange and add
010011e0 add     ecx,0x1
```

InterlockedIncrement compiles directly to machine code. The **lock xadd** instruction performs an atomic exchange and add. The final result is stored in **ecx**.

```
                    V_VT(&rgvarg[1]) = VT_I4;
                    V_I4(&rgvarg[1]) = fUnique ? lUnique : 0;

010011e3 mov     word ptr [rsp+0x98],0x3    ; V_VT(&rgvarg[1]) = VT_I4;
010011ed mov     eax,r14d                   ; rax = 0 (r14d is still zero)
010011f0 test    r12d,r12d                  ; fUnique set?
010011f3 cmovne  eax,ecx                    ; if so, then set rax=lCounter
010011f6 mov     [rsp+0xa0],eax             ; V_I4(&rgvarg[1]) = ...
```

Since x64 supports the **cmov** instruction, the **?:** construct can be compiled without using a jump.

```
                    dp.rgvarg = rgvarg;
                    dp.cArgs = 2;
                    dp.rgdispidNamedArgs = NULL;
                    dp.cNamedArgs = 0;

010011fd lea     rax,[rsp+0x80]             ; rax = &rgvarg[0]
01001205 mov     [rsp+0x60],rax             ; dp.rgvarg = rgvarg
0100120a mov     [rsp+0x70],r15d            ; dp.cArgs = 2 (r15 is still 2)
0100120f mov     [rsp+0x68],r14             ; dp.rgdispidNamedArgs = NULL
01001214 mov     [rsp+0x74],r14d            ; dp.cNamedArgs = 0
```

This code initializes the rest of the members of DISPPARAMS. Note that the compiler reuses the space on the stack previously used by the CLSID.

```
                    hr = pdisp->Invoke(dispid, IID_NULL, 0, DISPATCH_METHOD, &dp, NULL, NULL, NULL);

01001219 mov     rax,[rsi]                  ; rax = pdisp.vtbl
0100121c mov     [rsp+0x40],r14             ; arg9 = 0
01001221 mov     [rsp+0x38],r14             ; arg8 = 0
01001226 mov     [rsp+0x30],r14             ; arg7 = 0
0100122b lea     rcx,[rsp+0x60]             ; rcx = &dp
01001230 mov     [rsp+0x28],rcx             ; arg6 = &dp
01001235 mov     word ptr [rsp+0x20],0x1    ; arg5 = 1 (DISPATCH_METHOD)
0100123c xor     r9d,r9d                    ; arg4 = 0
0100123f lea     r8,[GUID_NULL (01001080)]  ; arg3 = &IID_NULL
01001246 mov     edx,r13d                   ; arg2 = dispid
01001249 mov     rcx,rsi                    ; arg1 = pdisp
0100124c call    qword ptr [rax+0x30]       ; call Invoke method
0100124f mov     ebx,eax                    ; hr = result
```

The code then sets up the parameters and calls the **Invoke** method.

```
                    VariantClear(&rgvarg[0]);
                    VariantClear(&rgvarg[1]);

01001251 lea     rcx,[rsp+0x80]             ; arg1 = &rgvarg[0]
01001259 call    qword ptr [_imp_VariantClear (01001018)]
0100125f lea     rcx,[rsp+0x98]             ; arg1 = &rgvarg[1]
01001267 call    qword ptr [_imp_VariantClear (01001018)]
0100126d jmp     ReleasePQA (01001274)
```

The code finishes up the current branch of the conditional, and skips over the **else** branch.

```
                } else {
                    hr = E_OUTOFMEMORY;
                }
            }

OutOfMemory:
0100126f mov     ebx,0x8007000e             ; hr = E_OUTOFMEMORY
        pqa->Release();
ReleasePQA:
01001274 mov     rcx,[rsp+0x58]             ; arg1 = pqa
01001279 mov     rax,[rcx]                  ; rax = pqa.vtbl
0100127c call    qword ptr [rax+0x10]       ; release
```

The **else** branch.

```
    return hr;
}

0100127f mov     eax,ebx                    ; rax = hr (for return value)
ReturnEAX:
01001281 add     rsp,0x2c0                  ; clean up the stack
01001288 pop     r15d                       ; restore
0100128a pop     r14d                       ; restore
0100128c pop     r13d                       ; restore
0100128e pop     r12d                       ; restore
01001290 pop     rdi                        ; restore
01001291 pop     rsi                        ; restore
01001292 pop     rbx                        ; restore
01001293 ret                                ; return (do not pop arguments)
```

The return value is stored in **rax**, and then the non-volatile registers are restored before returning.

Debugger Engine and Extension APIs

12/5/2018 • 3 minutes to read • Edit Online

This section includes:

Debugger Engine Overview

Using the Debugger Engine API

Writing DbgEng Extensions

EngExtCpp Extensions

Writing WdbgExts Extensions

Customizing Debugger Output Using DML

Using JavaScript to Extend the Capabilities of the Debugger

This documentation describes how to use the debugger engine and how to write extensions that will run in WinDbg, KD, CDB, and NTSD. These debugger extensions can be used when performing user-mode or kernel-mode debugging on Microsoft Windows.

Debugger Engine

The debugger engine provides an interface for examining and manipulating debugging targets in user-mode and kernel-mode on Microsoft Windows.

The debugger engine can acquire targets, set breakpoints, monitor events, query symbols, read and write memory, and control threads and processes in a target.

You can use the debugger engine to write both debugger extension libraries and stand-alone applications. Such applications are *debugger engine applications*. A debugger engine application that uses the full functionality of the debugger engine is a *debugger*. For example, WinDbg, CDB, NTSD, and KD are debuggers; the debugger engine provides the core of their functionality.

The debugger engine API is specified by the prototypes in the header file dbgeng.h.

For more information, see Debugger Engine Overview and Using the Debugger Engine API.

.

Extensions

You can create your own debugging commands by writing and building an extension DLL. For example, you might want to write an extension command to display a complex data structure.

There are three different types of debugger extension DLLs:

- *DbgEng extension DLLs*. These are based on the prototypes in the dbgeng.h header file. Each DLL of this type may export DbgEng extension commands. These extension commands use the Debugger Engine API and may also use the WdbgExts API.

  For more information, see Writing DbgEng Extensions.

- *EngExtCpp extension DLLs*. These are based on the prototypes in the engextcpp.h and dbgeng.h header files. Each DLL of this type may export DbgEng extension commands. These extension commands use both the Debugger Engine API and the EngExtCpp extension framework, and may also use the WdbgExts API.

- *WdbgExts extension DLLs*. These are based on the prototypes in the wdbgexts.h header file. Each DLL of this type exports one or more WdbgExts extension commands. These extension commands use the WdbgExts API exclusively. For more information see Writing WdbgExts Extensions.

The DbgEng API can be used to create extensions or stand-alone applications. The WdbgExts API contains a subset of the functionality of the debugger engine API and can be used only by extensions.

All debugger extensions should be compiled and built using Visual Studio.

Extension code samples are installed as part of the Debugging Tools for Windows package if you perform a custom installation and select the **SDK** component and all its subcomponents. They can be found in the sdk\samples subdirectory of the Debugging Tools for Windows installation directory.

The easiest way to write new debugger extensions is to study the sample extensions. Each sample extension includes makefile and sources files for use with the Build utility. Both types of extensions are represented in the samples.

Writing Custom Analysis Debugger Extensions

You can extend the capabilities of the **!analyze** debugger command by writing an analysis extension plugin. By providing an analysis extension plugin, you can participate in the analysis of a bug check or an exception in a way that is specific to your own component or application. When you write an analysis extension plugin, you also write a metadata file that describes the situations for which you want your plugin to be called. When **!analyze** runs, it locates, loads, and runs the appropriate analysis extension plugins. For more information, see Writing Custom Analysis Debugger Extensions

Customizing Debugger Output Using DML

You can customize debugger output using DML. For more information see Customizing Debugger Output Using DML.

Using JavaScript to Extend the Capabilities of the Debugger

Use JavaScript to create scripts that understand debugger objects and extend and customize the capabilities of the debugger. JavaScript providers bridge a scripting language to the debugger's internal object model. The JavaScript debugger scripting provider, allows the for use of JavaScript with the debugger. For more information, see JavaScript Debugger Scripting.

Debugger Engine Introduction

10/23/2019 • 3 minutes to read • Edit Online

This documentation describes how to use the debugger engine and how to write extensions that will run in WinDbg, KD, CDB, and NTSD. These debugger extensions can be used when performing user-mode or kernel-mode debugging on Microsoft Windows.

Debugger Engine

The debugger engine provides an interface for examining and manipulating debugging targets in user-mode and kernel-mode on Microsoft Windows.

The debugger engine can acquire targets, set breakpoints, monitor events, query symbols, read and write memory, and control threads and processes in a target.

You can use the debugger engine to write both debugger extension libraries and stand-alone applications. Such applications are *debugger engine applications*. A debugger engine application that uses the full functionality of the debugger engine is a *debugger*. For example, WinDbg, CDB, NTSD, and KD are debuggers; the debugger engine provides the core of their functionality.

The debugger engine API is specified by the prototypes in the header file dbgeng.h.

Incomplete Documentation

This is a preliminary document and is currently incomplete.

For many concepts relating to the debuggers and the debugger engine that are not yet documented here, look in the Debugging Techniques section of this documentation.

To obtain some of the currently undocumented functionality of the debugger engine API, use the **Execute** method to execute individual debugger commands.

Extensions

You can create your own debugging commands by writing and building an extension DLL. For example, you might want to write an extension command to display a complex data structure.

There are three different types of debugger extension DLLs:

- *DbgEng extension DLLs*. These are based on the prototypes in the dbgeng.h header file. Each DLL of this type may export DbgEng extension commands. These extension commands use the Debugger Engine API and may also use the WdbgExts API.
- *EngExtCpp extension DLLs*. These are based on the prototypes in the engextcpp.h and dbgeng.h header files. Each DLL of this type may export DbgEng extension commands. These extension commands use both the Debugger Engine API and the EngExtCpp extension framework, and may also use the WdbgExts API.
- *WdbgExts extension DLLs*. These are based on the prototypes in the wdbgexts.h header file. Each DLL of this type exports one or more WdbgExts extension commands. These extension commands use the WdbgExts API exclusively.

The DbgEng API can be used to create extensions or stand-alone applications. The WdbgExts API contains a subset of the functionality of the debugger engine API and can be used only by extensions.

All debugger extensions should be compiled and built by using the Build utility. The Build utility is included in the Windows Driver Kit (WDK).

Extension code samples are installed as part of the Debugging Tools for Windows package if you perform a custom installation and select the **SDK** component and all its subcomponents. They can be found in the sdk\samples subdirectory of the Debugging Tools for Windows installation directory.

The easiest way to write new debugger extensions is to study the sample extensions. Each sample extension includes makefile and sources files for use with the Build utility. Both types of extensions are represented in the samples.

Writing Custom Analysis Debugger Extensions

You can extend the capabilities of the **!analyze** debugger command by writing an analysis extension plugin. By providing an analysis extension plugin, you can participate in the analysis of a bug check or an exception in a way that is specific to your own component or application. When you write an analysis extension plugin, you also write a metadata file that describes the situations for which you want your plugin to be called. When **!analyze** runs, it locates, loads, and runs the appropriate analysis extension plugins. For more information, see Writing Custom Analysis Debugger Extensions

Customizing Debugger Output Using DML

You can customize debugger output using DML. For more information see Customizing Debugger Output Using DML.

Using JavaScript to Extend the Capabilities of the Debugger

Use JavaScript to create scripts that understand debugger objects and extend and customize the capabilities of the debugger. JavaScript providers bridge a scripting language to the debugger's internal object model. The JavaScript debugger scripting provider, allows the for use of JavaScript with the debugger. For more information, see JavaScript Debugger Scripting.

Debugger Engine Overview

10/23/2019 • 2 minutes to read • Edit Online

The *debugger engine* (DbgEng.dll), typically referred to as the *engine*, provides an interface for examining and manipulating debugging targets in *user mode* and *kernel mode* on Microsoft Windows.

The debugger engine can acquire targets, set breakpoints, monitor events, query symbols, read and write to memory, and control threads and processes in a target.

You can use the debugger engine to write both debugger extension libraries and stand-alone applications. Such applications are referred to as *debugger engine applications*. A debugger engine application that uses the full functionality of the debugger engine is called a *debugger*. For example, WinDbg, CDB, NTSD, and KD are debuggers; the debugger engine provides the core of their functionality.

Engine Concepts:

Debugging Session and Execution Model

Client Objects

Input and Output

Examining and Manipulating Targets:

Targets

Events

Breakpoints

Symbols

Memory

Threads and Processes

Incomplete Documentation

This is a preliminary document and is currently incomplete.

For many concepts relating to the debuggers and the debugger engine that are not yet documented here, look in the Debugging Techniques section of this documentation.

To obtain some of the currently undocumented functionality of the debugger engine API, use the **Execute** method to execute individual debugger commands.

Debugging Session and Execution Model

10/23/2019 • 2 minutes to read • Edit Online

The debugger engine can debug multiple targets, simultaneously. A *debugging session* begins when the engine acquires a target and continues until all of the targets have been discarded. A debugging session is *inaccessible* while the targets are executing and *accessible* when the current target is suspended. The engine can only be used to examine and manipulate targets while the session is accessible.

The main loop of a debugger typically consists of setting the execution status, calling the method **WaitForEvent** and handling the generated events. When **WaitForEvent** is called, the session becomes inaccessible.

When an event occurs in a target, the engine suspends all targets and the session becomes accessible. The engine then notifies the event callbacks of the event and follows the event filter rules. The event callbacks and event filters determine how execution in the target should proceed. If they determine that the engine should break into the debugger, the **WaitForEvent** method returns and the session remains accessible; otherwise, the engine will resume execution of the target in the manner determined by the event callbacks and event filters, and the session becomes inaccessible again.

For the duration of the **WaitForEvent** call--in particular, while notifying the event callbacks and processing the filter rules--the engine is in a state referred to as "inside a wait". While in this state, **WaitForEvent** cannot be called (it is not reentrant).

There are two steps involved in initiating execution in a target: setting the execution status, and then calling **WaitForEvent**. The execution status can be set using the method **SetExecutionStatus** or by executing a debugger command that sets the execution status--for example, **g(Go)** and **p (Step)**.

If a sequence of debugger commands are executed together--for example, "**g ; ? @$ip**"--an *implicit wait* will occur after any command that requires execution in the target if that command is not the last command in the sequence. An implicit wait cannot occur when the debugger engine is in the state "inside a wait"; in this case, the execution of the commands will stop and the current command--the one that attempted to cause the implicit wait--will be interpreted as an indication of how execution in the target should proceed. The rest of the commands will be discarded.

Note When determining whether the session is accessible or inaccessible, limited execution of a target (for example, stepping) is considered execution by the engine. When the limited execution is complete, the session becomes accessible.

Host Engine

When debugging remotely, you can use multiple instances of the debugger engine. Exactly one of these instances maintains the debugging session; this instance is called the *host engine*.

All debugger operations are relative to the host engine, for example, symbol loading and extension loading.

Client Objects

10/23/2019 • 2 minutes to read • Edit Online

Almost all interaction with the debugger engine is through *client objects*, often simply referred to as *clients*. Each client provides an implementation of the top-level engine interfaces. Each interface provides a different set of methods, which can be used to interact with the engine and, through the engine, the targets. An instance of the engine can have many clients, each with its own state.

Primary Clients

A *primary client* is a client that has joined the current debugging session. Initially, when a new client object is created, it is not a primary client. A client becomes a primary client when it is used to acquire a target (for example, by calling **CreateProcess2**) or is connected to the debugging session using **ConnectSession**. The debugger command **.clients** lists only the primary clients.

Callback Objects

Callback objects can be registered with each client. There are three types of callback objects:

1. **Input Callback Objects** (or *input callbacks*): the engine calls input callbacks to request input. For example, a debugger with a console window could register an input callback to provide the engine with input from the user, or a debugger might register an input callback to provide the engine with input from a file.
2. **Output Callback Objects** (or *output callbacks*): the engine calls output callbacks to display output. For example, a debugger with a console window could register an output callback to present the debugger's output to the user, or a debugger might register an output callback to send the output to a log file.
3. **Event Callback Objects** (or *event callbacks*): the engine calls event callbacks whenever an event occurs in a target (or there is a change in the engine's state). For example, a debugger extension library could register an event callback to monitor certain events or perform automated actions when an particular event occurs.

Remote Debugging

Client objects facilitate communication to remote instances of the host engine. The **DebugConnect** function creates a client object that is connected to a remote engine instance; methods called on this client are executed by the remote engine and callback objects registered locally with the client will be called when the remote engine makes callback calls.

Additional Information

For details about creating and using client objects, see Using Callback Objects. For details about registering callback objects, see Using Callback Objects.

Input and Output

10/23/2019 • 2 minutes to read • Edit Online

The input and output facilities of the debugger engine can be used for interactive debugger operation and logging. The input usually represents commands and responses that are typed by the user, and the output usually represents information presented to the user or sent to log files.

The debugger engine maintains an *input stream* and an *output stream*. Input can be requested from the input stream, and output sent to the output stream.

When the **Input** method is called to request input from the engine's input stream, the engine will call all the registered input callbacks to inform them that it is waiting for input. It then waits for the input callbacks to provide the input by calling the **ReturnInput** method.

When output is sent to the engine's output stream, the engine will call the registered output callbacks passing the output to them. When sending output to the output stream, it can be filtered by the client object; in which case, only output callbacks that are registered with particular client objects will receive the output.

The input and output streams are transparently available to the remote clients. Remote clients can request input and send output to the engine's input and output stream, and the engine will call the callbacks registered with remote clients to request input or send output.

Additional Information

For details about using input and output, see Using Input and Output. For more information about client objects and input and output callbacks, see Client Objects.

Remote Debugging (Debugger Engine)

10/23/2019 • 2 minutes to read • Edit Online

Remote debugging occurs when a client's communication with a target is indirect, for example, through a network connection. When remote debugging, more than one instance of the debugger engine can be involved in debugging a target. However, exactly one of these instances is responsible for the debugging session; this instance is called the *host engine*.

There are many possible configurations: the client object can be created in the host engine (smart clients), or a different instance of the engine (debugging clients); the host engine can be connected directly to the target (debugging server); or a proxy can be directly connected to the target (process server and kernel connection server).

Multiple clients can simultaneously connect to the host engine. And the host engine can connect to multiple targets in the same debugging session. Optionally, there can be one or more proxies between the clients and the host engine and between the host engine and each target.

Smart clients are client objects that communicate directly with the host engine. A debugging client is created by calling **DebugConnect**; the client communicates with the host engine using RPC calls that represent method calls in the engine's API (including calls that the host engine makes to the client's callback objects).

A debugging server is an engine instance that communicates directly with the target and is also the host engine. Process servers and kernel connection servers communicate directly with the target but are not the host engine. The host engine communicates with the process server, or kernel connection server, by sending low-level memory, processor, and operating system requests, and the server sends back the results.

Note A typical two-computer setup for kernel debugging--where one computer is the target and the other the host computer--is not considered to be remote debugging as there is only one instance of the engine (on the host computer) and it communicates directly with the target.

Additional Information

For details about performing remote debugging, see Remote Targets.

Targets

10/23/2019 • 2 minutes to read • Edit Online

The debugger engine supports debugging different types of targets, user-mode and kernel-mode targets, live targets and crash dump files, and local and remote targets. There are different methods for connecting the engine to these different types of targets.

Crash Dump Files

Both user-mode, and kernel-mode crash-dump files are opened with **OpenDumpFile**. The engine is also able to create dump files from a target with **WriteDumpFile2**.

Live, User-Mode Targets

The debugger engine can both create and attach to user-mode processes.

Creating a process is done by providing a command line, and optionally an initial directory and environment, for the new process. The engine can then connect to the new process, or keep the new process suspended while it connects to another process. For example, when debugging an application that consists of both a client and server, it is possible to create a client in a suspended state and attach to an already running server, allowing server breakpoints to be set before the client runs and provokes server operations.

When detaching from a process, the engine can optionally leave the process running normally, kill the process, or abandon the process (leaving it suspended until another debugger attaches to it or it is killed).

The engine can be queried for information about all of the user-mode processes that are running on the computer, including the process ID and name of the executable image that is used to start the process. This information can be used to help locate a process to debug.

Live, Kernel-Mode Targets

The method **AttachKernel** connects the debugger engine to a Windows kernel.

Remote Targets

When using the debugger engine to debug remotely, there are potentially two extra steps:

1. Connect to the host engine. If the host engine is not the local engine instance, use **DebugConnect** to create a client object that is connected to the host engine.
2. Connect the host engine to the process server or kernel connection server. If the host engine does not connect directly to the target, it must connect to a process server or kernel connection server that does.

Now the client can tell the host engine to acquire a target through the process server or kernel connection server.

Acquiring Targets

When acquiring a target, the acquisition of the target is not complete until the target generates an event. Typically, this means first calling a method to attach the debugger to the target, then calling **WaitForEvent** to let the target generate an event. This still holds true when the target is a crash dump file, as these always store an event-typically the event that caused the dump file to be created.

Additional Information

For details about attaching to targets, see Connecting to Targets.

Events

10/23/2019 • 2 minutes to read • Edit Online

The debugger engine provides facilities for monitoring and responding to events in the target. When an event occurs, the engine suspends the target (often only briefly), it then notifies all of the clients of the event, who in turn instruct the engine on how execution should proceed in the target.

To notify a client of an event, the engine calls the event callback object that is registered with the client. The engine provides each event callback with details of the event, and the event callback instructs the engine on how execution should proceed in the target. When different event callbacks provide conflicting instructions, the engine acts on the instruction with the highest precedence (see **DEBUG_STATUS_XXX**), which typically means choosing the instruction that involves the least execution of the target.

Note While the event callback is handling the event, the target is suspended and the debugging session is accessible; however, because the engine was waiting for an event--either explicitly during a **WaitForEvent** call or implicitly by executing a command such as **g (Go)** or **p (Step)**--the event callback cannot call **WaitForEvent**, and if it attempts to execute any commands that would cause the debugger to execute, for example **g (Go)** or **p (Step)**, the engine will interpret these commands as an instruction on how to proceed.

Event Filters

The debugger engine also provides *event filters*, which are a simpler alternative for basic event monitoring. The event filters provide a few simple rules that specify whether an event should be printed to the debugger's output stream or break into the debugger. They can also be used to execute debugger commands when an event occurs.

Additional Information

For details about monitoring events, see Monitoring Events.

Breakpoints

12/5/2018 • 2 minutes to read • Edit Online

The debugger engine can create and monitor breakpoints in the target.

There are two types of breakpoints that the engine can insert into a target: software breakpoints and processor breakpoints.

- *Software breakpoints* are inserted into the target's code by modifying the processor instruction at the breakpoint's location. The debugger engine keeps track of such breakpoints; they are invisible to the clients reading and writing memory at that location. A software breakpoint is triggered when the target executes the modified instruction.
- *Processor breakpoints* are inserted into the target's processor by the debugger engine. A processor breakpoint can be triggered by different actions, for example, executing an instruction at the location (like software breakpoints), or reading or writing memory at the breakpoint's location. Support for processor breakpoints is dependent on the processor in the target's computer.

A breakpoint's address can be specified by an explicit address, by an expression that evaluates to an address, or by an expression that might evaluate to an address at a future time. In the last case, each time a module is loaded or unloaded in the target, the engine will attempt to reevaluate the expression and insert the breakpoint if it can determine the address; this makes it possible to set breakpoints in modules before they are loaded.

A number of parameters can be associated with a breakpoint to control its behavior:

- A breakpoint can be associated with a particular thread in the target and will only be triggered by that thread.
- A breakpoint can have debugger commands associated with it; these commands will automatically be executed when the breakpoint is triggered.
- A breakpoint can be flagged as inactive until the target has passed it a specified number of times.
- A breakpoint can be automatically removed the first time it is triggered.

Additional Information

For details about using breakpoints, see Using Breakpoints.

Symbols

12/5/2018 • 2 minutes to read • Edit Online

A *symbol* is a named unit of data or code from a source file that appears in a module. Information about symbols can include the name, type (if applicable), the address or register where it is stored, and any parent or child symbols. Examples of symbols include variables (local and global), functions, and any entry point into a module.

The symbol information is used by the engine to help interpret data and code in the target. With this information, the engine can search for symbols by name or location in memory and provide a description of a symbol.

The engine gets its information about symbols from symbol files, which are located on the local file system or loaded from a symbol server. When using a symbol server, the engine will automatically use the correct version of the symbol file to match the module in the target. Symbol files can be loaded whenever the corresponding module is loaded, or they can be loaded as needed.

Note Often optimizing compilers do not include accurate information in symbol files. This can cause the engine to misinterpret the value of some variables as the variable's location or lifetime might be incorrectly described, causing the engine to look at the wrong piece of memory or think a variable value is live when it is dead (or vice versa). It is also possible for an optimizing compiler to change the order of execution or to split a function into several pieces. Best results are usually obtained when debugging unoptimized code.

Additional Information

For details about using symbols, see Using Symbols. For an overview of using symbol files and symbol servers, see Symbols in the Debuggers section of this documentation.

Memory

12/5/2018 • 2 minutes to read • Edit Online

The debugger engine can directly read and write the target's main memory, registers, and other data spaces. In kernel-mode debugging, all of the target's memory is available, including virtual memory, physical memory, registers, Model Specific Registers (MSRs), System Bus Memory, Control-Space Memory, and I/O Memory. In user-mode debugging, only the virtual memory and registers are available.

The engine exposes, to the clients, all memory in the target using 64-bit addresses. If the target uses 32-bit addresses, when communicating with the target and the clients, the engine will automatically convert between 32-bit and 64-bit addresses, as needed. If a 32-bit address is recovered from the target--for example, by reading from memory or a register--it must be sign-extended to 64 bits before it can be used in the debugger engine API. Sign extension is performed automatically by the **ReadPointersVirtual** method.

Additional Information

For details about reading and writing memory, see Memory Access.

Threads and Processes

12/5/2018 • 2 minutes to read • Edit Online

Terminology

Thread and a process concepts are different between user-mode debugging and kernel-mode debugging.

- In *user-mode debugging*, a *process* is an operating system process and a *thread* is an operating system thread.
- In *kernel-mode debugging*, the debugger engine creates a *virtual process* for each target; this process represents the kernel and does not correspond to any operating system process. For each physical processor in the target computer, the debugger creates a *virtual thread*; these threads represent the processors and do not correspond to any operating system threads.

When an event occurs, the engine sets the *event process* and *event thread* to the process and thread (operating system or virtual) in which the event occurred.

The *current thread* is the thread (operating system or virtual) that the engine is currently controlling. The *current process* is the process (operating system or virtual) that the engine is currently controlling. When an event occurs, the current thread and process are initially set to the event thread and process; but, they can be changed using the clients while the session is accessible.

In kernel mode, the debugger keeps track of an implicit process and implicit thread. The *implicit process* is the operating system process that determines the translation from virtual to physical memory addresses.

The *implicit thread* is the operating system thread that determines the target's registers, including call stack, stack frame, and instruction offset.

When an event occurs, the implicit thread and implicit process are initially set to the event thread and process; they can be changed while the session is accessible.

Thread and Process Data

The engine maintains several pieces of information about each thread and process. This includes the system thread and process ID and system handles, and the process environment (PEB), the thread environment block (TEB), and their locations in target's memory.

Additional Information

For details about using thread and processes, see Controlling Threads and Processes.

Using the Debugger Engine API

6/25/2019 • 2 minutes to read • Edit Online

This section includes:

Debugger Engine API Overview

Debugger Engine Reference

Debugger Engine API Overview

10/23/2019 • 2 minutes to read • Edit Online

This section includes:

Interacting with the Engine

Using Input and Output

Monitoring Events

Using Breakpoints

Memory Access

Examining the Stack Trace

Controlling Threads and Processes

Using Symbols

Using Source Files

Connecting to Targets

Target Information

Target State

Calling Extensions and Extension Functions

Assembling and Disassembling Instructions

Important The IDebug* interfaces such as **IDebugEventCallbacks** interface, although COM like, are not proper COM APIs. Calling these interfaces from managed code is an unsupported scenario. Issues such as garbage collection and thread ownership, lead to system instability when the interfaces are called with managed code.

Interacting with the Engine

10/23/2019 • 2 minutes to read • Edit Online

Commands and Expressions

The debugger engine API provides methods to execute commands and evaluate expressions, like those typed into WinDbg's Debugger Command Window. To execute a debugger command, use **Execute**. Or, to execute all of the commands in a file, use **ExecuteCommandFile**.

The method **Evaluate** will evaluate expressions using the C++ or MASM syntax. The syntax used by the debugger engine to evaluate expressions--such as in the **Evaluate** method--is given by **GetExpressionSyntax** and can be changed using **SetExpressionSyntaxByName** and **SetExpressionSyntax**. The number of different syntaxes that are recognized by the debugger is returned by **GetNumberExpressionSyntaxes**, and their names are returned by **GetExpressionSyntaxNames**.

The type of value that is returned by **Evaluate** is determined by the symbols and constants used in the string that was evaluated. The value is contained in a **DEBUG_VALUE** structure and can be cast to different types using **CoerceValue** and **CoerceValues**.

Aliases

Aliases are character strings that are automatically replaced with other character strings when used in debugger commands and expressions. For an overview of aliases, see Using Aliases. The debugger engine has several classes of aliases.

The *fixed-name aliases* are indexed by number and have the names **$u0**, **$u1**, ..., **$u9**. The values of these aliases can be set using the **SetTextMacro** method and can be retrieved using **GetTextMacro** method.

The *automatic aliases* and *user-named aliases* can have any name. The automatic aliases are defined by the debugger engine and the user-named aliases are defined by the user through debugger commands or the debugger engine API. To define or remove a user-named alias, use the **SetTextReplacement** method. The **GetTextReplacement** method returns the name and value of an automatic alias or a user-named alias. All the user-named aliases can be removed using the **RemoveTextReplacements** method. The **GetNumberTextReplacements** method will return the number of user-name and automatic aliases; this can be used with **GetTextReplacement** to iterate over all these aliases. The **OutputTextReplacements** method will print a list of all the user-named aliases, including their names and values.

Note if a user-named alias is given the same name as an automatic alias, the user-named alias will hide the automatic alias so that when retrieving the value of the alias by name, the user-named alias will be used.

Engine Options

The engine has a number of options that control its behavior. These options are listed in **DEBUG_ENGOPT_XXX**. They are returned by **GetEngineOptions** and can be set using **SetEngineOptions**. Individual options can be set using **AddEngineOptions** and unset using **RemoveEngineOptions**.

Interrupts

An interrupt is a way to force a break into the debugger or to tell the engine to stop processing the current command, for example, by pressing Ctrl+Break in WinDbg.

To request a break into the debugger, or to interrupt the debugger's current task, use **SetInterrupt**. To check if there has been an interrupt, use **GetInterrupt**.

Note When undertaking a long task from a debugger extension, it is recommended that the extension check **GetInterrupt** regularly and stop processing if an interrupt has been requested.

When requesting a break into the debugger, the engine might time out if it takes too long for the target to carry out the break-in. This can occur if the target is in a non-responsive state or if the break-in request is blocked or delayed by resource contention. The length of time the engine will wait is returned by **GetInterruptTimeout** and can be set using **SetInterruptTimeout**.

Using Client Objects

10/23/2019 • 2 minutes to read • Edit Online

For an overview of the role of client objects in interacting with the debugger engine, see Client Objects.

In general, a client's methods may be called only from the thread in which the client was created. Typically, methods called from the wrong thread will fail immediately. The notable exception to this rule is the method **CreateClient**; this method may be called from any thread, and returns a new client that can be used in the thread from which it was called. Other exceptions are documented in the reference section.

A string describing a client object is returned by the method **GetIdentity** or can be written to the engine's output stream using **OutputIdentity**.

COM Interfaces

The debugger engine API contains several COM like interfaces; they implement the **IUnknown** interface.

The interfaces described in the section Debug Engine Interfaces is implemented by the client (though not necessarily at the latest version). You may use the COM method **IUnknown::QueryInterface** to obtain each of these interfaces from any of the others.

The clients implement the **IUnknown** COM interface and use it for maintaining reference counts and interface selection. However, the clients are not registered COM objects. The method **IUnknown::AddRef** is used to increment the reference count on the object, and the method **IUnknown::Release** is used to decrement the reference count. When **IUnknown::QueryInterface** is called, the reference count is incremented, so when a client interface pointer is no longer needed **IUnknown::Release** should be called to decrement the reference count.

The reference count will be initialized to one when the client object is created using **DebugCreate** or **DebugConnect**.

See the Platform SDK for more information about when reference counts should be incremented and decremented.

IUnknown::QueryInterface, **DebugCreate**, and **DebugConnect** each take an interface ID as one of their arguments. This interface ID can be obtained using the **__uuidof** operator. For example:

```
IDebugClient * debugClient;
HRESULT Hr = DebugCreate( __uuidof(IDebugClient), (void **)&debugClient );
```

Important The IDebug* interfaces such as **IDebugEventCallbacks** interface, although COM like, are not proper COM APIs. Calling these interfaces from managed code is an unsupported scenario. Issues such as garbage collection and thread ownership, lead to system instability when the interfaces are called with managed code.

Using Callback Objects

10/23/2019 • 2 minutes to read • Edit Online

There are three callback COM like interfaces that are used by the engine: IDebugEventCallbacks for notifying debugger extensions and applications of changes to the engine or target, IDebugInputCallbacks for requesting input, and IDebugOutputCallbacks for sending output.

Callback objects are registered with clients. At most, one instance of each of the three callback interfaces can be registered with each client (the Unicode and ASCII versions of a interface count as the same interface).

When a client is created, the engine remembers the thread in which it was created. The engine uses this same thread whenever it makes a call to a callback instance registered with the client. If the thread is in use, the engine will queue the calls it needs to make. To allow the engine to make these calls, the method *DispatchCallbacks* should be called whenever a client's thread is idle. The method **ExitDispatch** will cause *DispatchCallbacks* to return. If the thread is the same thread that was used to start the debugger session, then the engine can make the callback calls during the **WaitForEvent** method, and *DispatchCallbacks* does not need to be called.

The method **FlushCallbacks** tells the engine to send all buffered output to the output callbacks.

Event Callback Objects

The IDebugEventCallbacks interface is used by the engine to notify the debugger extensions and applications of events and changes to the engine and target. An implementation of **IDebugEventCallbacks** can be registered with a client using *SetEventCallbacks*. The current implementation registered with a client can be found using *GetEventCallbacks*. The number of event callbacks registered across all clients can be found using *GetNumberEventCallbacks*.

For details on how the engine manages events, see Monitoring Events.

Input Callback Objects

The IDebugInputCallbacks interface is used by the engine to request input from debugger extensions and applications. An implementation of **IDebugInputCallbacks** can be registered with a client using *SetInputCallbacks*. The current implementation registered with a client can be found using *GetInputCallbacks*. The number of input callbacks registered across all clients can be found using *GetNumberInputCallbacks*.

For details on how the engine manages input, see Input and Output.

Output Callback Objects

The **IDebugOutputCallbacks** interface is used by the engine to send output to the debugger extensions and applications. An implementation of **IDebugOutputCallbacks** can be registered with a client using *SetOutputCallbacks*. The current implementation registered with a client can be found using *GetOutputCallbacks*. The number of output callbacks registered across all clients can be found using *GetNumberOutputCallbacks*.

For details on how the engine manages output, see Input and Output.

Note As is typical for COM objects, the engine will call **IUnknown::AddRef** on a callback COM object when it is registered with a client, and **IUnknown::Release** when the object is replaced or the client is deleted.

Using Input and Output

10/23/2019 • 3 minutes to read • Edit Online

For an overview of the input and output streams in the debugger engine, see Input and Output.

Input

The engine will ask for input from all its clients if the **Input** method is called on a client. The input is returned to the caller of **Input**.

Input Callbacks

When the engine asks for input from a client, it uses the IDebugInputCallbacks object registered with that client. An **IDebugInputCallbacks** object may be registered with a client using *SetInputCallbacks*. Each client can have at most one **IDebugInputCallbacks** object registered with it.

The request for input begins with the engine calling the **IDebugInputCallbacks::StartInput** method. This informs the **IDebugInputCallbacks** object that the engine is now waiting for input.

If the **IDebugInputCallbacks** object has some input for the engine, it can call the **ReturnInput** method of any client. Once the **ReturnInput** method has been called, the engine will not take any more input. Subsequent callers of this method will be informed that the input was not received.

The engine will then call **IDebugInputCallbacks::EndInput** to indicate that it has stopped waiting for input.

Finally, the engine will echo this input to the registered **IDebugOutputCallbacks** object of every client (except the one used to provide the input) by using **IDebugOutputCallbacks::Output** with the bit-mask set to DEBUG_OUTPUT_PROMPT.

Output

Output may be sent to the engine using several client methods -- for example *Output* and *OutputVaList*. Upon receiving output, the engine sends it to some clients.

Clients use an *output mask* to indicate which types of output they are interested in. Whenever output is produced by the engine, it is accompanied by a mask specifying its output type. If the type of output matches the output mask of the client, the client will receive the output. The output mask may be set by calling **SetOutputMask** and queried using **GetOutputMask**. See **DEBUG_OUTPUT_XXX** for details of the output mask values.

The list of clients that the engine will send output to is controlled by the *output control*. Typically, the output control is set to send output to all clients; however, it can be temporarily changed using *ControlledOutput* and *ControlledOutputVaList*. See **DEBUG_OUTCTL_XXX** for details about output control values.

Output may be buffered by the engine. If multiple pieces of output are passed to the engine, it may collect them and send them to the clients in one large piece. To flush this buffer, use **FlushCallbacks**.

Each client object has an *output width*, which is the width of the output display for the client object. While this width is only used as a hint, some commands and extension functions format their output based on this width. The width is returned by the GetOutputWidth method and can be set using the SetOutputWidth method.

Output Callbacks

When the engine sends output to a client, it uses the IDebugOutputCallbacks object registered with the client. An **IDebugOutputCallbacks** object may be registered with a client using *SetOutputCallbacks*. Each client can have at most one **IDebugInputCallbacks** object registered with it.

To send the output, the engine calls the **IDebugOutputCallbacks::Output** method.

Output Line Prefix

Each client object has an *output line prefix* which is prepended to every line of output sent to the output callback associated with the client object. This can be used for indentation or to place identifying marks on each line of output.

The output line prefix is returned by GetOutputLinePrefix and can be set using SetOutputLinePrefix. To temporarily change the output line prefix and later change it back again, use PushOutputLinePrefix and PopOutputLinePrefix.

Log Files

The debugger engine supports opening a log file to record a debugging session. At most, one log file can be open at a time. Output sent to the output callbacks is also sent to this log file (unless it is flagged to not be logged).

To open a log file, use **OpenLogFile2** (or **OpenLogFile**). The method **GetLogFile2** (or **GetLogFile**) returns the currently open log file. To close the log file, use **CloseLogFile**.

The method **SetLogMask** can be used to filter the output sent to the log file, and **GetLogMask** will return the current log file filter.

Prompt

In an interactive debugging session, a prompt can be used to indicate to the user that the debugger is waiting for user input. The prompt is sent to the output callbacks using the *OutputPrompt* and *OutputPromptVaList* methods. The contents of the standard prompt are returned by **GetPromptText**.

Monitoring Events

10/23/2019 • 2 minutes to read • Edit Online

For an overview of events in the debugger engine, see Events.

Events occurring in a target or the debugger engine may be monitored using the IDebugEventCallbacks interface. An **IDebugEventCallbacks** object may be registered with a client using *SetEventCallbacks*. Each client can only have at most one **IDebugEventCallbacks** object registered with it.

When an **IDebugEventCallbacks** object is registered with a client, the engine will call the object's **IDebugEventCallbacks::GetInterestMask** to determine which events the object is interested in. Only events in which the object is interested will be sent to it.

For each type of event, the engine calls a corresponding callback method on IDebugEventCallbacks. For events from the target, the **DEBUG_STATUS_XXX** value returned from these calls specifies how the execution of the target should proceed. The engine collects these return values from each **IDebugEventCallbacks** object it calls and acts on the one with the highest precedence.

Events from the Target That Break into the Debugger by Default

The following events break into the debugger by default:

- Breakpoint Events
- Exception Events (not documented here)
- System Error

Events from the Target that Do Not Break into the Debugger by Default

The following events do not break into the debugger by default:

- Create Process Event
- Exit Process Event
- Create Thread Event
- Exit Thread Event
- Load Module Event
- Unload Module Event

Internal Engine Changes

The following are not actual events, but are merely internal engine changes:

- Target Change
- Engine Change
- Engine Symbol Change
- Session Status Change

Event Filters

10/23/2019 • 4 minutes to read • Edit Online

Event filters provide simple event filtering; they influence how the debugger engine proceeds after an event occurs in a target. When an event occurs, the engine determines whether that event matches an event filter. If it does, the break status for the event filter influences whether the debugger will break into the target. If the event is an exception event, the handling status determines whether the exception should be considered handled or not-handled in the target.

Note If more sophisticated event filtering is required, event callbacks can be used.

Event filters are divided into three categories.

1. *Specific event filters*. These are the filters for all the non-exception events. See **DEBUG_FILTER_XXX** for a list of these events.
2. *Specific exception filters*. The first specific exception filter is the *default exception filter*. The rest are filters for those exceptions for which the engine has built-in filters. See **Specific Exceptions** for a list of the specific exception filters.
3. *Arbitrary exception filters*. These are filters for exception events that have been added manually.

The filters in categories 1 and 2 are collectively known as *specific filters*, and the filters in categories 2 and 3 are collectively known as *exception filters*. The number of filters in each category is returned by **GetNumberEventFilters**.

An event matches a specific event filter if the type of the event is the same as the type of the filter. Some event filters have an additional parameter which further restricts the events they match.

An exception event matches an exception filter if the exception code for the exception event is the same as the exception code for the exception filter. If there is no exception filter with the same exception code as the exception event, the exception event will be handled by the default exception filter.

Commands and Parameters

Event filters can have a debugger command associated with them. This command is executed by the engine when an event matching the filter occurs. **GetEventFilterCommand** and **SetEventFilterCommand** can be used to get and set this command. For exception filters, this command is executed on the first-chance of the exception. A separate second-chance command can be executed upon the second-chance exception event. To get and set the second-chance command, use **GetExceptionFilterSecondCommand** and **SetExceptionSecondChanceCommand**.

The parameters for specific event filters and exception filters are returned by **GetSpecificFilterParameters** and **GetExceptionFilterParameters**. The break status and handling status for event filters can be set using **SetSpecificFilterParameters** and **SetExceptionFilterParameters**.

SetExceptionFilterParameters can also be used to add and remove arbitrary exception filters.

A short description of specific filters is returned by **GetEventFilterText**.

Some specific filters take arguments that restrict which events the filter matches. **GetSpecificFilterArgument** and **SetSpecificFilterArgument** will get and set arguments for those specific filters which support arguments. If a specific filter has no argument, there is no restriction on which events it matches. The following table lists the event filters that take arguments and how they restrict the events which match them:

EVENT	MATCH CRITERIA
Create Process	The name of the created process must match the argument.1
Exit Process	The name of the exited process must match the argument.1
Load Module	The name of the loaded module must match the argument.1
Unload Module	The base address of the unloaded module must be the same as the argument.2
Target Output	The debug output from the target must match the argument.3

Note

1. The argument uses the string wildcard syntax and is compared with the image name (ignoring path) when the event occurs. If the name of the module or process is not available, it is considered a match.
2. The argument is an expression that is evaluated by the engine when the argument is set.
3. The argument uses the string wildcard syntax and is compared with the debug output from the target. If the output is not known, it is considered a match.

Index and Exception Code

Each event filter has an index. The index is a number between zero and one less than the total number of filters (inclusive). The index range for each category of filters can be found from the *SpecificEvents*, *SpecificExceptions*, and *ArbitraryExceptions* values returned by **GetNumberEventFilters**, as described in the following table:

EVENT FILTERS	INDEX OF FIRST FILTER	NUMBER OF FILTERS
Specific event filters	0	*SpecificEvents*
specific exception filters	*SpecificEvents*	*SpecificExceptions*
arbitrary exception filters	*SpecificEvents* + *SpecificExceptions*	*ArbitraryExceptions*

The indices for the specific event filters are found in the first table located in the topic **DEBUG_FILTER_XXX**. The index of the default exception filter (the first specific exception filter) is *SpecificEvents*. When an arbitrary exception filter is removed, the indices of the other arbitrary exception filters can change.

The exception filters are usually specified by exception code. However, some methods require the index of the exception. To find the index of an exception filter for a given exception, use **GetExceptionFilterParameters** to iterate over all the exception filters until you find the one with the same exception code as the exception. The exception codes for the specific exception filters can be found in the topic **Specific Exceptions**.

System Errors

When a system error occurs, the engine will break into the debugger or print the error to the output stream, if the error occurs at or below specified levels. These levels are returned by **GetSystemErrorControl** and can be changed using **SetSystemErrorControl**.

Event Information

10/23/2019 • 2 minutes to read • Edit Online

Whenever a debugging session is accessible, there is a *last event*. This is the event that caused the session to become accessible. The *event target* is the target which generated the last event. When the session becomes accessible, the current target is set to the event target. The details of the last event are returned by **GetLastEventInformation**. The instruction pointer for the last event and the memory at the instruction pointer when the event occurred are returned by the **Request** operations **DEBUG_REQUEST_GET_CAPTURED_EVENT_CODE_OFFSET** and **DEBUG_REQUEST_READ_CAPTURED_EVENT_CODE_STREAM**.

If the target is a crash dump file, the *last event* is the last event that occurred before the dump file was created. This event is stored in the dump file and the engine generates it for the event callbacks when the dump file is acquired as a debugging target.

If the target is a kernel-mode target and a *bug check* occurred, the bug check code and related parameters can be found using **ReadBugCheckData**.

If the target is a user-mode Minidump, the dump file generator may store an additional event. Typically, this is the event that provoked the generator to save the dump file. Details of this event are returned by **GetStoredEventInformation** and the **Request** operations **DEBUG_REQUEST_TARGET_EXCEPTION_CONTEXT**, **DEBUG_REQUEST_TARGET_EXCEPTION_THREAD**, and **DEBUG_REQUEST_TARGET_EXCEPTION_RECORD**.

Dump files may contain a static list of events. Each event represents a snapshot of the target at a particular point in time. The number of events in this list is returned by **GetNumberEvents**. For a description of each event in the list, use **GetEventIndexDescription**. To set an event from this list as the current event, use the method **SetNextEventIndex**; after calling **WaitForEvent**, the event becomes the current event. To determine which event in the list is the current event, use **GetCurrentEventIndex**.

Using Breakpoints with the Debugger Engine API

12/5/2018 • 2 minutes to read • Edit Online

Breakpoints are event triggers which, when the breakpoint's conditions are satisfied, will halt execution of the target and break into the debugger. Breakpoints allow the user to analyze and perhaps modify the target when execution reaches a certain point or when a certain memory location is accessed.

The debugger engine inserts a *software breakpoint* into a target by modifying the processor instruction at the breakpoint's location; this modification is invisible to the engine's clients. A software breakpoint is triggered when the target executes the instruction at the breakpoint location. A *processor breakpoint* is inserted into the target's processor by the debugger engine; its capabilities are processor-specific. It is triggered by the processor when the memory at the breakpoint location is accessed; what type of access will trigger this breakpoint is specified when the breakpoint is created.

This topic includes:

Setting Breakpoints

Controlling Breakpoint Flags and Parameters

Setting Breakpoints

10/23/2019 • 2 minutes to read • Edit Online

Breakpoints are created with the **AddBreakpoint** method. This method creates an IDebugBreakpoint object that represents the breakpoint. It also set the *breakpoint type* (software breakpoint or processor breakpoint). Once a breakpoint has been created, its type cannot be changed.

Breakpoints are deleted with the **RemoveBreakpoint** method. This also deletes the **IDebugBreakpoint** object; this object may not be used again.

Note Although **IDebugBreakpoint** implements the **IUnknown** interface, the methods **IUnknown::AddRef** and **IUnknown::Release** are not used to control the lifetime of the breakpoint. These methods have no effect on the lifetime of the breakpoint. Instead, an **IDebugBreakpoint** object is deleted after the method **RemoveBreakpoint** is called.

When the breakpoint is created, it is given a unique *breakpoint ID*. This identifier will not change. However, after the breakpoint has been deleted, its ID may be used for another breakpoint. For details on how to receive notification of the removal of a breakpoint, see Monitoring Events.

When a breakpoint is created, it is initially disabled; this means that it will not cause the target to stop executing. This breakpoint may be enabled by using the method **AddFlags** to add the DEBUG_BREAKPOINT_ENABLED flag.

When a breakpoint is first created, it has the memory location 0x00000000 associated with it. The location can be changed by using **SetOffset** with an address, or by using **SetOffsetExpression** with a symbolic expression. The breakpoint's location should be changed from its initial value before it is used.

Controlling Breakpoint Flags and Parameters

10/23/2019 • 4 minutes to read • Edit Online

There are a number of methods that can be used to determine basic information about breakpoints:

- **GetId** returns the breakpoint ID.
- **GetType** returns the breakpoint type (software or processor) and the type of the effective processor on which the breakpoint is set.
- **GetAdder** returns the client that added the breakpoint.
- **GetOffset** returns the address of a breakpoint.
- **GetOffsetExpression** returns the expression string that specifies the location of the breakpoint.

In addition to its location and breakpoint type, a breakpoint has several parameters controlling its behavior.

Breakpoint parameters can be controlled through a variety of specific methods. In addition, most of the parameters may be queried together using **GetParameters**.

Breakpoint Flags

Breakpoint flags are one kind of breakpoint parameters.

Breakpoint flags can be queried using **GetFlags**. They can be changed by using **AddFlags**, **RemoveFlags**, or **SetFlags**.

Breakpoint flags form a bit field. The possible flags that can be used in this bit field, and their meanings, are as follows:

DEBUG_BREAKPOINT_ENABLED
When this flag is set, the breakpoint is *enabled* and will have its normal effect. When this flag is not set, the breakpoint is *disabled* and will not have any effect. If you wish to temporarily deactivate a breakpoint, you can remove this flag; it is then easy to add this flag back when you want to re-enable this breakpoint.

DEBUG_BREAKPOINT_ADDER_ONLY
When this flag is set, the breakpoint is a *private breakpoint*. This breakpoint is visible only to the client that added it. In this case, other clients will not be able to query the engine for the breakpoint, and the engine will not send events generated by the breakpoint to other clients. All callbacks (event and output) related to this breakpoint will be sent only to this client. See **GetAdder**.

DEBUG_BREAKPOINT_GO_ONLY
When this flag is set, the breakpoint will only be triggered if the target is in unrestricted execution. It will not be triggered if the engine is stepping through instructions in the target.

DEBUG_BREAKPOINT_ONE_SHOT
When this flag is set, the breakpoint will automatically remove itself the first time it is triggered.

DEBUG_BREAKPOINT_DEFERRED
When this flag is set, the breakpoint is *deferred*. This flag is set by the engine when the offset of the breakpoint is specified using a symbolic expression, and the engine cannot evaluate the expression. Every time a module is loaded or unleaded in the target, the engine will attempt reevaluate the expression for all breakpoints whose location is specified using an expression. Those that cannot be evaluated are flagged as deferred. This flag cannot be modified by any client.

Other Breakpoint Parameters

Breakpoint parameters also include:

Pass count
If the breakpoint has a pass count associated with it, it will not be activated until the target has passed the breakpoint the specified number of times. The pass count that was originally set can be found by using **GetPassCount**. The number of times remaining that the engine will pass the breakpoint before it is activated can be found using **GetCurrentPassCount**. The pass count can be reset to a new value by using **SetPassCount**.

Match thread
If the breakpoint has a thread associated with it, it will be ignored by the engine when it is encountered by any other thread. The thread can be found by using **GetMatchThreadId**, and can be changed by using **SetMatchThreadId**.

Command
The breakpoint may have a command associated with it. The command is executed when the breakpoint is activated. This command can be found by using **GetCommand**, and can be changed by using **SetCommand**.

Size
If the breakpoint is a processor breakpoint, it must have a specified size. This determines the size of the block of memory whose access will activate the breakpoint -- the beginning of the block is the breakpoint's location. The size can be found by using **GetDataParameters**, and can be changed by using **SetDataParameters**.

Access type
If the breakpoint is a processor breakpoint, it must have an access type. This determines the type of access that will activate the breakpoint. For example, the breakpoint may be activated if the target reads from, writes to, or executes the memory specified by the breakpoint. The access type can be found by using **GetDataParameters**, and can be changed by using **SetDataParameters**.

Valid Parameters for Processor Breakpoints

The following access types are available for processor breakpoints:

VALUE	DESCRIPTION
DEBUG_BREAK_READ	The breakpoint will be triggered when the CPU reads memory in the breakpoint's memory block.
DEBUG_BREAK_WRITE	The breakpoint will be triggered when the CPU writes memory in the breakpoint's memory block.
DEBUG_BREAK_READ | DEBUG_BREAK_WRITE	The breakpoint will be triggered when the CPU reads or writes memory in the breakpoint's memory block.
DEBUG_BREAK_EXECUTE	The breakpoint will be triggered when the CPU fetches the instruction in the breakpoint's memory block.
DEBUG_BREAK_IO	The breakpoint will be triggered when the I/O port in the breakpoints memory block is accessed. (Windows XP and Microsoft Windows Server 2003 only, kernel mode only, x86 only)

Not all access types and sizes are supported on all processors. The following access types and sizes are supported:

x86
All access types are supported. DEBUG_BREAK_READ behaves like DEBUG_BREAK_READ | DEBUG_BREAK_WRITE. The size must be 1, 2, or 4. The breakpoint's address must be a multiple of the size.

x64
All access types are supported. DEBUG_BREAK_READ behaves like DEBUG_BREAK_READ | DEBUG_BREAK_WRITE. The size must be 1, 2, 4, or 8. The breakpoint's address must be a multiple of the size.

Itanium
All access types except DEBUG_BREAK_IO are supported. The size must be a power of two; for DEBUG_BREAK_EXECUTE, the size must be 16. The breakpoint's address must be a multiple of the size.

Itanium running in x86 mode
The is the same as for x86, except that DEBUG_BREAK_IO is not supported.

Memory Access

1/11/2019 • 2 minutes to read • Edit Online

The debugger engine to directly read from and write to the target's main memory, registers, and other data spaces.

In user-mode debugging, only the virtual memory and registers can be accessed; the physical memory and other data spaces cannot be accessed.

The debugger engine API always uses 64-bit addresses when referring to memory locations on the target.

If the target uses 32-bit addresses, the native 32-bit address will automatically be sign-extended by the engine to 64-bit addresses. The engine will automatically convert between 64-bit addresses and native 32-bit addresses when communicating with the target.

This section includes:

Virtual and Physical Memory

Registers

Other Data Spaces

Virtual and Physical Memory

10/23/2019 • 2 minutes to read • Edit Online

The engine provides a number of methods for reading and writing the virtual and physical memory of a target.

Virtual Memory

When specifying a location in the virtual memory of a target, the target's virtual address space is used. In user-mode debugging, this is the virtual address space of the current process. In kernel-mode debugging, this is the virtual address space of the implicit process. See Threads and Processes for more information about the current and implicit process.

The virtual memory (of the target) can be read by using **ReadVirtual** and written using **WriteVirtual**.

Pointers in the target's memory can be read and written by using the convenience methods **ReadPointersVirtual** and **WritePointersVirtual**. These methods will automatically convert between the 64-bit pointers used by the engine and the native pointers used by the target. These methods are useful when requesting memory that contains pointers that will be used for subsequent requests -- for example, a pointer to a string.

The **SearchVirtual** and **SearchVirtual2** methods can be used to search the target's virtual memory for a pattern of bytes.

The **FillVirtual** method can be used to copy a pattern of bytes, multiple times, to the target's virtual memory.

The target's virtual memory can also be read and written in a way that bypasses the debugger engine's virtual memory cache using the methods **ReadVirtualUncached** and **WriteVirtualUncached**. These uncached versions are useful for reading virtual memory that is inherently volatile, such as memory-mapped device areas, without contaminating or invalidating the cache. Uncached memory access should only be used in situations when it is required, as the performance of uncached access can be significantly lower than cached access.

The engine provides some convenience methods to read strings from the target's virtual memory. To read a multibyte string from the target, use **ReadMultiByteStringVirtual** and **ReadMultiByteStringVirtualWide**. To read a Unicode string from the target, use **ReadUnicodeStringVirtual** and **ReadUnicodeStringVirtualWide**.

To find information about a memory location, use **GetOffsetInformation**. Not all of the virtual address space in the target contains valid memory. To find valid memory within a region, use **GetValidRegionVirtual**. When manually searching for valid memory in a target, the method **GetNextDifferentlyValidOffsetVirtual** will find the next location where the validity may change.

Physical Memory

The physical memory can only be directly accessed in kernel-mode debugging.

Physical memory on the target can be read by using **ReadPhysical** and **ReadPhysical2**, and written by using **WritePhysical** and **WritePhysical2**.

The **FillPhysical** method can be used to copy a pattern of bytes, multiple times, to the target's physical memory.

An address in the target's virtual address space can be translated to a physical address on the target by using the **VirtualToPhysical** method. The system's paging structures used to translate a virtual address to a physical address can be found by using **GetVirtualTranslationPhysicalOffsets**.

Events

When the virtual or physical memory of the target is changed, the **IDebugEventCallbacks::ChangeDebuggeeState** callback method is called.

Registers

10/23/2019 • 2 minutes to read • Edit Online

The debugger engine can be used to examine and alter the target's registers.

The registers available on the target depend on its processor architecture. For a description of the registers for the x86 and Itanium processors, see Processor Architecture. For a complete description of the registers available for a processor, see that processor's documentation.

The Register Set

The **GetNumberRegisters** method can be used to find the number of registers on the target.

Each register is referred to by its index. The index of the first register is zero, and the index of the last register is the number of registers minus one. To find the index of a register whose name is known, use **GetIndexByName**.

The method **GetDescription** returns information about a register. This includes the register's name, the type of values it can hold, and whether it is a subregister.

A *subregister* is a register that is contained within another register. When the subregister changes, the register that contains it also changes. For example, on an x86 processor, the **ax** subregister is the same as the low 16 bits of the 32-bit **eax** register.

There are three special registers whose values may be found by using the following methods. The interpretation of the values of these registers is platform dependent.

- The location of the current instruction may be found with **GetInstructionOffset** and **GetInstructionOffset2**.
- The location of the current processor stack slot may be found with **GetStackOffset** and **GetStackOffset2**.
- The location of the stack frame for the current function may be found with **GetFrameOffset** and **GetFrameOffset2**.

Manipulating Registers

The value of a register can be read by using the method **GetValue**. Multiple registers can be read by using **GetValues** and **GetValues2**.

A value can be written to a register by using the method **SetValue**. Multiple registers can be written by using **SetValues** and **SetValues2**.

When writing a value to a register, if the value supplied has a different type to the type of the register then the value is converted into the register's type. This conversion is the same as that performed by the method **CoerceValue**. This conversion may result in data loss if the register's type is not capable of holding the value supplied.

Pseudo-Registers

Pseudo-registers are variables maintained by the debugger engine that hold certain values, for example, **$teb** is the name of the pseudo-register whose value is the address of the current thread's Thread Environment Block (TEB). For more information, and a list of the pseudo-registers, see Pseudo-Register Syntax.

Each pseudo-register has an index. The index is a number between zero and the number of pseudo-registers - (returned by **GetNumberPseudoRegisters**) minus one. To find the index of a pseudo-register by its name, use **GetPseudoIndexByName**. The values of pseudo-registers can be read using **GetPseudoValues**, and values can be written to pseudo-registers using **SetPseudoValues**. For a description of a pseudo-register, including its type,

use **GetPseudoDescription**.

Note Not all of the pseudo-registers are available in all debugging sessions or at all times in a particular session.

Displaying Registers

The methods **OutputRegisters** and **OutputRegisters2** format the target's registers and sends them to the clients as output.

Events

Whenever the values of the target's registers change, the engine will call the **IDebugEventCallbacks::ChangeDebuggeeState** callback method with the parameter *Flags* set to DEBUG_CDS_REGISTERS.

Other Data Spaces

10/23/2019 • 2 minutes to read • Edit Online

In kernel-mode debugging, it is possible to read and write data to a variety of data spaces in addition to the main memory and registers. The following data spaces can be accessed:

System Bus
The methods **ReadBusData** and **WriteBusData** read and write system bus data.

Control-Space Memory
The methods **ReadControl** and **WriteControl** read and write control-space memory.

I/O Memory.
The methods **ReadIo** and **WriteIo** read and write system and bus I/O memory.

Model Specific Register (MSR)
The methods **ReadMsr** and **WriteMsr** read and write MSRs, which are control registers that enable and disable features, and support debugging, for a particular model of CPU.

Handles

In user-mode debugging, information about system objects can be obtained using system handles owned by a target process. The method **ReadHandleData** can be used to read this information.

System handles for thread and process system objects can be obtained by using the **GetCurrentThreadHandle** and **GetCurrentProcessHandle** methods. These handles are also provided to the **IDebugEventCallbacks::CreateThread** and **IDebugEventCallbacks::CreateProcess** callback methods when create-thread and create-process debugging event occur.

Note In kernel mode, the process and thread handles are artificial handles. They are not system handles.

Examining the Stack Trace

10/23/2019 • 2 minutes to read • Edit Online

A *call stack* contains the data for the functions calls made by a thread. The data for each function call is called a *stack frame* and includes the return address, parameters passed to the function, and the function's local variables. Each time a function call is made a new stack frame is pushed onto the top of the stack. When that function returns, the stack frame is popped off the stack.

Each thread has its own call stack, representing the calls made in that thread.

To get a stack trace, use the methods **GetStackTrace** and **GetContextStackTrace**. A stack trace can be printed using **OutputStackTrace** and **OutputContextStackTrace**.

Controlling Threads and Processes

10/23/2019 • 6 minutes to read • Edit Online

For an overview of threads and processes in the debugger engine, see Threads and Processes.

When an event occurs, the event thread and event process are set to the thread and process (operating system or virtual) in which the event occurred. They can be found using **GetEventThread** and **GetEventProcess**, respectively.

Implicit Threads and Processes

In kernel-mode debugging the debugger engine will use the *implicit process* to determine which virtual address space to use when performing virtual to physical address translation -- for example, in the methods **VirtualToPhysical** and **ReadVirtual**. When an event occurs, the implicit process is set to the current process.

The implicit process may be changed by using **SetImplicitProcessDataOffset**. To determine the implicit process use **GetImplicitProcessDataOffset**.

Note When setting breakpoints during a live kernel debugging session, the debugger engine will pass the virtual address of the breakpoint to the target, and the target will set the breakpoint. In this case, only the process context of the target is used when handling the breakpoint; the value of the implicit process is irrelevant.

In kernel-mode debugging, the debugger engine will use the *implicit thread* to determine some of the target's registers. This includes the processor stack (see **GetStackOffset**), the frame offset (see **GetFrameOffset**), and the instruction offset (see **GetInstructionOffset**). When an event occurs, the implicit thread is set to the current thread.

The implicit thread may be changed by using **SetImplicitThreadDataOffset**. To determine the implicit thread, use **GetImplicitThreadDataOffset**.

Not all registers are determined by the implicit thread. Some registers will remain the same when the implicit thread is changed.

Warning The implicit process and implicit thread are independent. If the implicit thread does not belong to the implicit process, then user and session state for the implicit thread will be in the wrong virtual address space and attempts to access this information will cause errors or provide incorrect results. This problem does not occur when accessing kernel memory, since kernel memory addresses are constant across all virtual address spaces. Thus information for the implicit thread located in kernel memory may be accessed independent of the implicit process.

Threads

The *engine thread ID* is used by the debugger engine to identify each operating system thread and each virtual thread for a target.

While a target is stopped, each thread also has an index relative to the process to which it belongs. For any process, the index of the first thread in the process is zero, and the index of the last thread is the number of threads in the process minus one. The number of threads in the current process can be found by using **GetNumberThreads**. The total number of threads in all processes in the current target can be found by using **GetTotalNumberThreads**.

The engine thread ID and system thread ID for one or more threads in the current process can be found from their index by using **GetThreadIdsByIndex**.

The engine maintains several pieces of information about each thread. This information may be queried for the

current thread, and may be used to find the engine thread ID for a thread.

system thread ID (user-mode debugging only)
The system thread ID of the current thread can be found by using **GetCurrentThreadSystemId**. For a given system thread ID, the corresponding engine thread ID may be found by using **GetThreadIdBySystemId**.

thread environment block (TEB)
The address of the TEB for the current thread can be found by using **GetCurrentThreadTeb**. For a given TEB address, the corresponding engine thread ID may be found by using **GetThreadIdByTeb**. In kernel-mode debugging, the TEB of a (virtual) thread is the TEB of the system thread that was running on the corresponding processor when the last event occurred.

data offset
In user-mode debugging, the data offset of a (system) thread is the location of the TEB for that thread. In kernel-mode debugging the data offset of a (virtual) thread is the KTHREAD structure for the system thread that was running on the corresponding processor when the last event occurred. The data offset of the current thread can be found by using **GetCurrentThreadDataOffset**. For a given data offset, the corresponding engine thread ID may be found by using **GetThreadIdByDataOffset**.

system handle
The system handle of the current thread can be found by using **GetCurrentThreadHandle**. For a given system handle, the corresponding engine thread ID may be found by using **GetThreadIdByHandle**. In kernel-mode debugging, an artificial handle is created for each (virtual) process. This handle can only be used with debugger engine API queries.

Processes

The *engine process ID* is used by the debugger engine to identify each operating system process and each virtual process for a target.

While a target is stopped, each process has an index relative to the target. The index of the first process in the target is zero, and the index of the last process is the number of processes in the target minus one. The number of processes in the current target can be found by using **GetNumberProcesses**.

The engine process ID and system process ID for one or more threads in the current target can be found from their index by using **GetProcessIdsByIndex**.

The engine maintains several pieces of information about each process. This information may be queried for the current process, and may be used to find the engine process ID for a process.

system process ID (user-mode debugging only)
The system process ID of the current process can be found by using **GetCurrentProcessSystemId**. For a given system process ID, the corresponding engine process ID may be found by using **GetProcessIdBySystemId**.

process environment block (PEB)
The address of the PEB for the current process can be found by using **GetCurrentProcessPeb**. For a given PEB address, the corresponding engine process ID may be found by using **GetProcessIdByPeb**. In kernel-mode debugging, the PEB of the (virtual) process is the PEB of the system process that was running when the last event occurred.

data offset
In user-mode debugging, the data offset of a (system) process is the location of the PEB of that process. In kernel-mode debugging, the data offset of the (virtual) process is the KPROCESS structure for the system process that was running when the last event occurred. The data offset of the current process can be found by using **GetCurrentProcessDataOffset**. For a given data offset, the corresponding engine process ID may be found by using **GetProcessIdByDataOffset**.

system handle

The system handle of the current process can be found by using **GetCurrentProcessHandle**. For a given system handle, the corresponding engine process ID may be found by using **GetProcessIdByHandle**. In kernel-mode debugging, an artificial handle is created for the (virtual) process. This handle can only be used with debugger engine queries.

Events

In live user-mode debugging, whenever a thread is created or exits in a target, the create-thread and exit-thread debugging events are generated. These events result in calls to the **IDebugEventCallbacks::CreateThread** and **IDebugEventCallbacks::ExitThread** callback methods.

In live user-mode debugging, whenever a process is created or exits in a target, the create-process and exit-process debugging events are generated. These events result in calls to the **IDebugEventCallbacks::CreateProcess** and **IDebugEventCallbacks::ExitProcess** callback methods.

For more information about events, see Monitoring Events.

Additional Information

For more information about threads and processes, including the TEB, KTHREAD, PEB, and KPROCESS structures, see *Microsoft Windows Internals* by David Solomon and Mark Russinovich.

Using Symbols

10/23/2019 • 2 minutes to read • Edit Online

For an overview of symbols, including using symbol files and symbol servers, see Symbols.

Symbol Names and Locations

To find the location of a symbol given its name, use **GetOffsetByName**. For details on the syntax used to specify symbol names, see Symbol Syntax and Symbol Matching.

If the exact name of a symbol is not known, or multiple symbols have the same name, **StartSymbolMatch** will begin a search for symbols whose names match a given pattern. For details on the syntax, see String Wildcard Syntax.

To find the name of a symbol given its location, use **GetNameByOffset**. To find the names of symbols in a module near a given location, use **GetNearNamebyOffset**.

Note Whenever possible, qualify the symbol with the module name -- for example **mymodule!main**. Otherwise, if the symbol does not exist (for example, because of a typographical error) the engine will have to load and search the symbols for every module; this can be a slow process, especially for kernel-mode debugging. If the symbol name was qualified with a module name, the engine will only need to search the symbols for that module.

A symbol is uniquely identified using the structure **DEBUG_MODULE_AND_ID**. This structure is returned by the methods **GetSymbolEntriesByName** and **GetSymbolEntriesByOffset**, which search for symbols based on their name and location, respectively.

The method **GetSymbolEntryInformation** returns a description of a symbol using the **DEBUG_SYMBOL_ENTRY** structure.

To find the offset of a field within a structure, use **GetFieldOffset**. To find the name of a field given its index within a structure, use **GetFieldName**. To find the name of an enumeration constant given its value, use **GetConstantName**.

The method **GetSymbolInformation** can perform several requests for information about symbols.

Symbol Options

A number of options control how the symbols are loaded and unloaded. For a description of these options, see Setting Symbol Options.

Symbol options may be turned on by using **AddSymbolOptions**, and turned off by using **RemoveSymbolOptions**.

GetSymbolOptions returns the current symbol options. To set all the symbol options at once, use **SetSymbolOptions**.

Reloading Symbols

After loading symbol files, the engine stores the symbol information in an internal cache. To flush this cache, use **Reload**. These symbols will have to be loaded again now or at a later time.

Synthetic Symbols

Synthetic symbols are a way to label an arbitrary address for easy reference. Synthetic symbols can be created in any existing module. The method **AddSyntheticSymbol** creates a new synthetic symbol. Synthetic symbols can be removed using **RemoveSyntheticSymbol**. Reloading the symbols for the module deletes all synthetic symbols associated with that module.

Symbol Path

To add a directory or symbol server to the symbol path, use the method **AppendSymbolPath**. The whole symbol path is returned by **GetSymbolPath** and can be changed using **SetSymbolPath**.

Modules

10/23/2019 • 2 minutes to read • Edit Online

An *image* is an executable, DLL, or driver that Windows has loaded as part of a user-mode process or the kernel. The file from which the image was loaded is referred to as its *image file*.

The debugger engine caches a list of *modules* for each process (or, in kernel-mode, the virtual process). Typically each module in this list represents an image in the process. The engine's module list can be synchronized with the target using **Reload**.

Note In kernel-mode debugging, the engine's module list for the virtual process contains both the system-wide kernel-mode modules and the current process's user-mode modules.

A module can be specified by its base address in the target's virtual address space, or by its index in the list of modules the engine maintains for the target. The module's index equals its position in the list of modules, and therefore this index will change if a module with a lower index is unloaded. All unloaded modules have indexes; these are always higher than the indexes of loaded modules. The base address of a module will not change as long as it remains loaded; in some cases it may change if the module is unloaded and then reloaded.

The index is a number between zero and the number of modules in the target minus one. The number of modules in the current process can be found by calling **GetNumberModules**.

The index can be used to find the base address by calling **GetModuleByIndex**. The base address of a module owning a symbol with a given name can be found using **GetSymbolModule**.

The following methods return both the index and base address of the specified module:

- To find a module with a given module name, use **GetModuleByModuleName**.
- The module whose virtual address range contains a given address is returned by **GetModuleByOffset**. This method can be used to find the module index given the base address of the module.

The following methods return information about modules specified either by base address or index:

- The names of a module are returned by **GetModuleNames** and **GetModuleNameString**.
- Version information for the module is returned by **GetModuleVersionInformation**.
- Some of the parameters used to describe a module are returned by **GetModuleParameters**. For details on the parameters returned by this method, see **DEBUG_MODULE_PARAMETERS**.

Unloaded Modules

During user-mode debugging, unloaded modules are tracked only in Windows Server 2003 and later versions of Windows. Earlier versions of Windows only tracked unloaded modules in kernel mode. When they are tracked, they are indexed after the loaded modules. Hence any method that searches the target's modules will search all the loaded modules and then the unloaded modules. Unloaded modules can be used to analyze failures caused by an attempt to call unloaded code.

Synthetic Modules

Synthetic modules can be created as a way to label a region of memory. These modules cannot contain real symbols, but they can contain synthetic symbols. The method **AddSyntheticModule** creates a new synthetic module. Synthetic modules can be removed using **RemoveSyntheticModule**. Reloading all the modules in the target deletes all synthetic modules.

Image Path

The *executable image path* is used by the engine when searching for executable images.

The executable image path can consist of several directories separated by semicolons (;). These directories are searched in order.

For an overview of the executable image path, see Executable Image Path.

To add a directory to the executable image path, use the method **AppendImagePath**. The whole executable image path is returned by **GetImagePath** and can be changed using **SetImagePath**.

Additional Information

For more information about processes and virtual processes, see Threads and Processes.

Types

10/23/2019 • 2 minutes to read • Edit Online

Type information from a module's symbol file is identified by two pieces of information: a type ID and the base address of the module to which the type belongs. The following methods can be used to find a type ID:

- **GetTypeId** returns the type ID for a given type name.
- **GetSymbolTypeId** returns the type ID for the type of a symbol with the given name.
- **GetOffsetTypeId** returns the type ID for the symbol found at the given location.

The name and size of a type are returned by **GetTypeName** and **GetTypeSize**, respectively.

The following convenience methods can be used for reading and writing typed data in the target's physical and virtual memory:

ReadTypedDataPhysical

WriteTypedDataPhysical

ReadTypedDataVirtual

WriteTypedDataVirtual

Printing Typed Data

To format typed data and send it to the output callbacks, use *OutputTypedDataPhysical* and *OutputTypedDataVirtual* for data in the target's physical and virtual memory respectively.

The type options described in **DEBUG_TYPEOPTS_XXX** affect how the engine formats typed data before sending it to the output callbacks.

The type options may be turned on by using **AddTypeOptions**, and turned off by using **RemoveTypeOptions**.

GetTypeOptions returns the current type options. To set all the type options at once, use **SetTypeOptions**.

Interpreting Raw Data Using Type Information

The debugger engine API supports interpreting typed data. This provides a way to walk object hierarchies on the target, including finding members of structures, dereferencing pointers, and locating array elements.

Typed data is described by instances of the **DEBUG_TYPED_DATA** structure and represents regions of memory on the target cast to a particular type. The **DEBUG_REQUEST_EXT_TYPED_DATA_ANSIRequest** operation is used to manipulate these instances. They can be initialized to the result of expressions or by casting regions of memory to a specified type. For a list of all the sub-operations that the DEBUG_REQUEST_EXT_TYPED_DATA_ANSI **Request** operation supports, see **EXT_TDOP**.

Additional Information

For details on output callbacks, see Input and Output.

Scopes and Symbol Groups

10/23/2019 • 4 minutes to read • Edit Online

A *symbol group* contains a set of symbols for efficient manipulation as a group. A symbol group can be created and populated manually or can be automatically generated and updated based on symbols in lexical scopes, such as local variables and function arguments. The interface IDebugSymbolGroup is used to represent a symbol group.

There are two ways to create a symbol group. An empty symbol group is returned by **CreateSymbolGroup**, and the symbol group for the current lexical scope is returned by **GetScopeSymbolGroup**.

Note The symbol group generated from the current scope is a snapshot of the local variables. If any execution occurs in the target, the symbols may no longer be accurate. Also, if the current scope changes, the symbol group will no longer represent the *current* scope (because it will continue to represent the scope for which it was created).

Symbols can be added to a symbol group using **AddSymbol**, and removed using **RemoveSymbolByIndex** or **RemoveSymbolByName**. The method **OutputAsType** tells the debugger to use a different symbol type when handling a symbol's data.

Note The values for scoped symbols may not be accurate. In particular, the machine architecture and compiler optimizations may prevent the debugger from accurately determining a symbol's value.

The *symbol entry information* is a description of a symbol, including its location and its type. To find this information for a symbol in a module, use the **IDebugSymbols3::GetSymbolEntryInformation**. To find this information for a symbol in a symbol group, use **IDebugSymbolGroup2::GetSymbolEntryInformation**. See **DEBUG_SYMBOL_ENTRY** for details of the symbol entry information.

The following methods return information about a symbol in a symbol group:

- **GetSymbolName** returns the name of the symbol.
- **GetSymbolOffset** returns the absolute address in the target's virtual address space of the symbol, if the symbol has an absolute address.
- **GetSymbolRegister** returns the register containing the symbol, if the symbol is contained in a register.
- **GetSymbolSize** returns the size of the data for the symbol.
- **GetSymbolTypeName** returns the name of the symbol's type.
- **GetSymbolValueText** returns the value of the symbol as a string.

If a symbol is stored in a register or in a memory location known to the debugger engine, its value can be changed using **WriteSymbol**.

A symbol is a *parent symbol* if it contains other symbols. For example, a structure contains its members. A symbol is a *child symbol* if it is contained in another symbol. A symbol may be both a parent and child symbol. Each symbol group has a flat structure and contains parent symbols and their children. Each symbol has a *depth* -- symbols without parents in the symbol group have a depth of zero, and the depth of each child symbol is one greater than the depth of its parent. The children of a parent symbol may or may not be present in the symbol group. When the children are present in the symbol group, the parent symbol is referred to as *expanded*. To add or remove the children of a symbol in a symbol group, use **ExpandSymbol**.

The number of symbols in a symbol group is returned by **GetNumberSymbols**. The *index* of a symbol in a symbol group is an identification number; the index ranges from zero to the number of symbols minus one. Each time a symbol is added to or removed from a symbol group -- for example, by expanding a symbol -- the index of

all the symbols in the symbol group may change.

The symbol parameters, including information about parent-child relationships, can be found by using **GetSymbolParameters**. This method returns a **DEBUG_SYMBOL_PARAMETERS** structure.

The symbols in a symbol group can be printed to the debugger's output stream using the method **OutputSymbols**.

Scopes

The *current scope*, or *current local context*, determines the local variables exposed by the debugger engine. The scope has three components:

1. A stack frame.
2. A current instruction.
3. A register context.

If the stack frame is at the top of the call stack, the current instruction is the instruction that resulted in the last event. Otherwise the current instruction is the function call which resulted in the next higher stack frame.

The methods **GetScope** and **SetScope** can be used to get and set the current scope. When an event occurs, the current scope is set to the scope of the event. The current scope can be reset to the scope of the last event using **ResetScope**.

Thread Context

The *thread context* is the state preserved by Windows when switching threads. This is similar to the register context, except that there is some kernel-only processor state that is part of the register context but not the thread context. This extra state is available as registers during kernel-mode debugging.

The thread context is represented by the CONTEXT structure defined in ntddk.h. This structure is platform-dependent and its interpretation depends on the effective processor type. The methods **GetThreadContext** and **SetThreadContext** can be used to get and set the thread context.

Using Source Files

10/23/2019 • 2 minutes to read • Edit Online

The debugger engine maintains a *source path*, which is a list of directories and source servers that contain source code files associated with the current targets. The debugger engine can search these directories and source servers for the source files. With the help of symbol files, the debugger engine can match lines in the source files with locations in the target's memory.

For an overview of using source files with debuggers, see Debugging in Source Mode. For an overview of source paths, see Source Path. For an overview of using source servers from the debugger engine, see Using a Source Server.

Source Path

To add a directory or source server to the source path, use the method **AppendSourcePath**. The whole source path is returned by **GetSourcePath** and can be changed using **SetSourcePath**. A single directory or source server can be retrieved from the source path using **GetSourcePathElement**.

To find a source file relative to the source path, use **FindSourceFile** or, for more advanced options when using source servers, use **FindSourceFileAndToken**. **FindSourceFileAndToken** can also be used along with **GetSourceFileInformation** to retrieve variables related to a file on a source server.

Matching Source Files to Code in Memory

The debugger engine provides three methods for locating the memory locations that correspond to lines in a source file. To map a single line of source code to a memory location, use **GetOffsetByLine**. To search for memory locations for more than one source line or for nearby source lines, use **GetSourceEntriesByLine**. The **GetSourceFileLineOffsets** method will return the memory location of every line in a source file.

To perform the opposite operation and find the line of a source file that matches a location in the target's memory, use **GetLineByOffset**.

Note The relationship between memory locations and lines in a source file is not necessarily one-to-one. It is possible for a single line of source code to correspond to multiple memory locations and for a single memory location to correspond to multiple lines of source code.

Connecting to Targets

12/5/2018 • 2 minutes to read • Edit Online

This section includes:

Live User-Mode Targets

Live Kernel-Mode Targets

Dump-File Targets

Remote Targets

Live User-Mode Targets

10/23/2019 • 2 minutes to read • Edit Online

The methods for creating and attaching to processes that are listed in this topic can be used for the local computer and for a remote computer running a process server.

A user-mode process can be created using **Create Process** or **CreateProcess2**, which execute a given command to create a process. The method **AttachProcess** can be used to attach the debugger engine to an existing user-mode process. **CreateProcessAndAttach** and **CreateProcessAndAttach2** create a new user-mode process and attach to it or another user-mode process on the same computer. The **Request** operations **DEBUG_REQUEST_GET_ADDITIONAL_CREATE_OPTIONS**, **DEBUG_REQUEST_SET_ADDITIONAL_CREATE_OPTIONS**, and **DEBUG_REQUEST_SET_LOCAL_IMPLICIT_COMMAND_LINE** can be used to set some of the default options for creating processes.

Note The engine doesn't completely attach to the process until the **WaitForEvent** method has been called. Only after the process has generated an event -- for example, the process creation event -- does it become available in the debugger session. See Debugging Session and Execution Model for more details.

The method **GetRunningProcessSystemIds** will return the process IDs of all the running processes on the computer. The process ID of a particular program can be found using **GetRunningProcessSystemIdByExecutableName**. Given a process ID, a description of the process is returned by **GetRunningProcessDescription**.

Process Options

The process options determine part of the engine's behavior when attached to a user-mode process, including whether or not the debugger engine will automatically attach to child processes created by the target process and what the engine does with the target processes when it exits. See **DEBUG_PROCESS_XXX** for a description of the process options.

The process options can be queried using **GetProcessOptions**. They can be changed using **AddProcessOptions**, **RemoveProcessOptions**, and **SetProcessOptions**.

Disconnecting from Processes

There are three different ways for the engine to disconnect from a process.

1. *Detach*. Resume all the threads in the process so that it will continue running, no longer being debugged. **DetachCurrentProcess** will detach the engine from the current process and **DetachProcesses** will detach the engine from all processes. Not all targets support detaching. The **Request** operation **DEBUG_REQUEST_TARGET_CAN_DETACH** can be used to check if the target supports detaching.

2. *Terminate*. Attempt to kill the process. **TerminateCurrentProcess** will terminate the current process and **TerminateProcesses** will terminate all processes in the debugger session.

3. *Abandon*. Remove the process from the list of processes being debugged. The operating system will still consider the process as being debugged and it will remain suspended until another debugger attaches to it or it is killed. **AbandonCurrentProcess** will abandon the current process.

Live Kernel-Mode Targets

10/23/2019 • 2 minutes to read • Edit Online

To attach the debugger engine to a target computer for kernel-mode debugging, use the method **AttachKernel**.

Note The engine doesn't completely attach to the kernel until the **WaitForEvent** method has been called. Only after the kernel has generated an event -- for example, the initial breakpoint -- does it become available in the debugger session. See Debugging Session and Execution Model for more details.

If the debugger engine is attached to a kernel which is not the local kernel and the connection is not an eXDI connection, the connection options used to find the target computer can be queried using **GetKernelConnectionOptions**. The connection can also be re-synchronized or the connection speed changed using **SetKernelConnectionOptions**.

The debugger can attach to the local kernel, but only in a limited way and only if the computer was booted with the **/debug** boot switch. (In some Windows installations, local kernel debugging is supported when other switches are used, such as **/debugport**, but this is not a guaranteed feature of Windows and should not be relied on.) **IsKernelDebuggerEnabled** is used to determine if the local computer is available for debugging. For more information about kernel debugging on a single machine, see Performing Local Kernel Debugging.

Dump-File Targets

10/23/2019 • 2 minutes to read • Edit Online

For an introduction and overview of crash dump files, see Crash Dump Files.

Opening Dump Files

To open a crash dump file for use as a debugger target, use **OpenDumpFile** or **OpenDumpfileWide**. These methods are similar to the **.opendump** debugger command.

Note The engine doesn't completely attach to the dump file until the **WaitForEvent** method has been called. When a dump file is created from a process or kernel, information about the last event is stored in the dump file. After the dump file is opened, the next time execution is attempted, the engine will generate this event for the event callbacks. Only then does the dump file become available in the debugging session. See Debugging Session and Execution Model for more details.

Additional files can be used to assist in debugging a crash dump file. The methods **AddDumpInformationFile** and **AddDumpInformationFileWide** register files containing page-file information to be used when the next dump file is opened. These methods must be called before the dump file is opened. **GetNumberDumpFiles** will return the number of such files that were used when the current dump file was opened and **GetDumpFile** will return a description of these files.

User-mode minidump files contain several streams of information. These streams can be read using the **Request** operation **DEBUG_REQUEST_READ_USER_MINIDUMP_STREAM**.

Creating Dump Files

To create a crash dump file of the current target -- user-mode or kernel-mode -- use **WriteDumpFile2**. This method is similar to the **.dump** debugger command.

Remote Targets

10/23/2019 • 2 minutes to read • Edit Online

There are two different forms of remote debugging, depending on which computer (remote client or server) is the host computer. The *host computer* is the computer on which the debugger engine is active. On the other computer, the debugger engine is merely acting as a proxy relaying commands and data to the host engine.

All debugger operations -- such as executing commands and extensions, and symbol loading -- are performed by the host engine. A debugger session is also relative to the host engine.

To list the debugging servers and process servers currently running on a computer, use **OutputServers**.

Debugging Servers and Debugging Clients

A *debugging server* is an instance of the debugger engine acting as a host and listening for connections from debugging clients. The method **StartServer** will tell the debugger engine to start listening for connections from debugging clients.

A *debugging client* is an instance of the debugger engine acting as a proxy, sending debugger commands and I/O to the debugging server. The function **DebugConnect** can be used to connect to the debugging server.

The client object returned by **DebugConnect** is not automatically joined to the debugger session on the debugging server. The method **ConnectSession** can be used to join the session, synchronizing input and output.

The communication between a debugging server and a debugging client mostly consists of debugger commands and RPC calls sent to the server, and command output sent back to the client.

Process Servers, Kernel Connection Servers, and Smart Clients

Process servers and *kernel connection servers* are both instances of the debugger engine acting as proxies, listening for connections from smart clients, and performing memory, processor, or operating system operations as requested by these remote clients. A *process server* facilitates the debugging of processes that are running on the same computer. A *kernel connection server* facilitates the debugging of a Windows kernel debugging target that is connected to the computer that is running the connection server. A process server can be started using the method **StartProcessServer** or the program DbgSrv. The method **WaitForProcessServerEnd** will wait for a process server started with **StartProcessServer** to end. A kernel connection server can be activated using the program **KdSrv**.

A *smart client* is an instance of the debugger engine acting as a host engine and connected to a process server. The method **ConnectProcessServer** will connect to a process server. Once connected, the methods described in Live User-Mode Targets can be used.

When the remote client is finished with the process server, it can disconnect using **DisconnectProcessServer**, or it can use **EndProcessServer** to request that the process server shut down. To shut down the process server from the computer that it is running on, use Task Manager to end the process. If the instance of the debugger engine that used **StartProcessServer** is still running, it can use **Execute** to issue the debugger command **.endsrv 0**, which will end the process server (this is an exception to the usual behavior of **.endsrv**, which generally does not affect process servers).

The communication between a process server and a smart client typically consists of low-level memory, processor, and operating system operations and requests that are sent from the remote client to the server. Their results are then sent back to the client.

Target Information

10/23/2019 • 2 minutes to read • Edit Online

The method **GetDebuggeeType** returns the nature of the current target (for example, whether it is a kernel-mode or user-mode target), and how the debugger engine is connected to it.

If the target is a crash dump file file, the method **GetDumpFormatFlags** will indicate what information is contained in the dump.

Target's Computer

The page size of the target's computer is returned by **GetPageSize**. **IsPointer64Bit** will indicate if the computer uses 32-bit or 64-bit addresses.

Note Internally, the debugger engine always uses 64-bit addresses for the target. If the target only uses 32-bit addresses, the engine automatically converts them when communicating with the target.

The number of processors in the target's computer is returned by **GetNumberProcessors**.

There are three different processor types associated with the target's computer:

- The *actual processor type* is the type of the physical processor in the target's computer. This is returned by **GetActualProcessorType**.
- The *executing processor type* is the type of the processor used in the currently executing processor context. This is returned by **GetExecutingProcessorType**.
- The *effective processor type* is the processor type the debugger uses when interpreting information from the target -- for example, setting breakpoints, accessing registers, and getting stack traces. The effective processor type is returned by **GetEffectiveProcessorType** and can be changed using **SetEffectiveProcessorType**.

The effective processor type and executing processor type may differ from the actual processor type -- for example, when the physical processor is an x64 processor and it is running in x86 mode.

The different executing processor types that are supported by the physical processor on the target's computer are returned by **GetPossibleExecutingProcessorTypes**. The number of these is returned by **GetNumberPossibleExecutingProcessorTypes**.

The list of processor types that is supported by the debugger engine is returned by **GetSupportedProcessorTypes**. The number of supported processor types is returned by **GetNumberSupportedProcessorTypes**.

The names (full and abbreviated) of a processor type are returned by **GetProcessorTypeNames**.

The current time on the target's computer is returned by **GetCurrentTimeDate**. The length of time the target's computer has been running since the last boot is returned by **GetCurrentSystemUpTime**. Time information may not be available for all targets.

Target Versions

The Windows version running on the target's computer is returned by **GetSystemVersionValues** and the **Request** operation **DEBUG_REQUEST_GET_WIN32_MAJOR_MINOR_VERSIONS**, and a description of the Windows version is returned by **GetSystemVersionString**. Some of this information is also returned by **GetSystemVersion**.

Target State

10/23/2019 • 2 minutes to read • Edit Online

The method **OutputCurrentState** will print the current state of the target to the debugger's output stream.

The current execution status of the target is returned by **GetExecutionStatus**. If the target is suspended, the method **SetExecutionStatus** can be used to resume execution in one of the execution modes.

The method **GetReturnOffset** returns the address of the instruction that will execute when the current function returns.

GetNearInstruction returns the location of an instruction relative to a given address.

Examining the Stack Trace

A *call stack* contains the data for the function calls that are made by a thread. The data for each function call is called a *stack frame* and includes the return address, parameters passed to the function, and the function's local variables. Each time a function call is made, a new stack frame is pushed onto the top of the stack. When that function returns, the stack frame is popped off the stack. Each thread has its own call stack, which represents the calls that are made in that thread.

Note Not all of the data for a function call can be stored in the stack frame. Parameters and local variables, at times, can be stored in registers.

To retrieve the call stack or *stack trace*, use the methods **GetStackTrace** and **GetContextStackTrace**. The stack trace can be printed using **OutputStackTrace** and **OutputContextStackTrace**.

Calling Extensions and Extension Functions

10/23/2019 • 2 minutes to read • Edit Online

To load an extension library (or to obtain a handle for an already loaded extension library), use **AddExtension**. An extension library can be unloaded with **RemoveExtension**.

Extension commands can be called using **CallExtension**.

Extension Functions

Extension functions are functions that are exported by extension libraries. They can use any function prototype and are called directly using C function pointers.

They are not extension commands and are not available via debugger commands. Extension functions cannot be called remotely; they must be called directly. Hence they cannot be used from debugging clients. They can only be called when the client object is inside the host engine - when not remotely debugging or when using a smart client.

Extension functions are identified within extension libraries by the "_EFN_" prepended to their names.

To obtain a pointer to an extension function, use **GetExtensionFunction**. The type of this function pointer should match the prototype of the extension function. The extension function can now be called just like any other function pointer in C.

Example

If the following extension function was included in an extension library and loaded into the debugger engine:

```
HRESULT CALLBACK
_EFN_GetObject(IDebugClient * client, SomeObject * obj);
```

It could be called using:

```
typedef ULONG (CALLBACK * GET_OBJECT)(IDebugClient * client, SomeObject * obj);



HRESULT status = S_OK;
GET_OBJECT extFn = NULL;
SomeObject myObj;

if (g_DebugControl->
        GetExtensionFunction(0,
                             "GetObject",
                             (FARPROC *) &extFn ) == S_OK &&
    extFn)
{
    status = (*extFn)(client, &myObj);
}
```

Assembling and Disassembling Instructions

10/23/2019 • 2 minutes to read • Edit Online

The debugger engine supports the use of assembly language for displaying and changing code in the target. For an overview of the use of assembly language in the debugger, see Debugging in Assembly Mode.

Note Assembly language is not supported for all architectures. And on some architectures not all instructions are supported.

To assemble a single assembly-language instruction and place the resulting processor instruction in the target's memory, use **Assemble**.

To disassemble a single instruction by taking a processor instruction from the target and producing a string that represents the assembly instruction, use **Disassemble**.

The method **GetDisassembleEffectiveOffset** returns the first effective address of the last instruction to be disassembled. For example, if the last instruction to be disassembled is `move ax, [ebp+4]`, the effective address is the value of `ebp+4`. This corresponds to the **$ea** pseudo-register.

To send disassembled instructions to the output callbacks, use the methods *OutputDisassembly* and *OutputDisassemblyLines*.

The debugger engine has some options that control the assembly and disassembly. These options are returned by **GetAssemblyOptions**. They can be set using **SetAssemblyOptions** and some options can be turned on with **AddAssemblyOptions** or turned off with **RemoveAssemblyOptions**.

Writing DbgEng Extensions

6/25/2019 • 2 minutes to read • Edit Online

This section includes:

DbgEng Extension Design Guide

DbgEng Extension Reference

DbgEng Extension Design Guide

12/5/2018 • 2 minutes to read • Edit Online

This section includes:

Anatomy of a DbgEng Extension DLL

Using Clients and the Engine

Writing DbgEng Extension Code

Building DbgEng Extensions

Anatomy of a DbgEng Extension DLL

10/23/2019 • 2 minutes to read • Edit Online

A DbgEng extension DLL exports a number of callback functions, some of which may be implementations of extension commands.

These extension DLLs are loaded by the debugger engine and can provide extra functionality or automation of tasks while performing user-mode or kernel-mode debugging on Microsoft Windows.

If you performed a full install of Debugging Tools for Windows, a sample DbgEng extension called "exts" can be found in the sdk\samples\exts subdirectory of the installation directory.

Extension Commands

An extension DLL may export any number of functions that are used to execute extension commands. Each function is explicitly declared as an export in the .def file, and its name must consist entirely of lowercase letters.

Functions used to implement extension commands must match the prototype **PDEBUG_EXTENSION_CALL**.

These functions are named according to the standard C++ convention, except that uppercase letters are not permitted. The exported function name and the extension command name are identical, except that the extension command begins with an exclamation point (!). For example, when you load myextension.dll into the debugger and then type **!stack** into the Debugger Command window, the debugger looks for an exported function named **stack** in myextension.dll.

If myextension.dll is not already loaded, or if there may be other extension commands with the same name in other extension DLLs, you can type **!myextension.stack** into the Debugger Command window to indicate the extension DLL and the extension command in that DLL.

Other Exported Functions

A DbgEng extension DLL must export *DebugExtensionInitialize*. This will be called when the DLL is loaded, to initialize the DLL. It may be used by the DLL to initialize global variables.

An extension DLL may export *DebugExtensionUninitialize*. If this is exported, it will be called before the extension DLL is unloaded. It may be used by the DLL to clean up before it is unloaded.

An extension DLL may export *DebugExtensionNotify*. If this is exported, it will be called when a session begins or ends, and when a target starts or stops executing. These notifications are also provided to IDebugEventCallbacks objects registered with a client.

An extension DLL may export *KnownStructOutput*. If this is exported, it will be called when the DLL is loaded. This function returns a list of structures that the DLL knows how to print on a single line. It may be called later to format instances of these structures for printing.

Engine Procedure for Loading a DbgEng Extension DLL

When an extension DLL is loaded, the callback functions are called by the engine in the following order:

1. **DebugExtensionInitialize** is called so the extension DLL can initialize.
2. If exported, **DebugExtensionNotify** is called if the engine has an active session, and called again if the session is suspended and accessible.
3. If exported, **KnownStructOutput** is called to request a list of structures the DLL knows how to print on a single line.

See Loading Debugger Extension DLLs for information about how to use the debugger to load and unload an extension DLL, and see Using Debugger Extension Commands for information about executing an extension command.

The debugger engine will place a **try / except** block around a call to an extension DLL. This protects the engine from some types of bugs in the extension code; but, since the extension calls are executed in the same thread as the engine, they can still cause it to crash.

Using Clients and the Engine

10/23/2019 • 2 minutes to read • Edit Online

A DbgEng extension interacts with the debugger engine through a client object.

When an extension function is called, it is passed a client. The extension function should use this client for all its interaction with the debugger engine, unless it has a specific reason to use another client.

An extension library may create its own client object upon initialization by using **DebugCreate**. This client can be used to register callback objects from the DLL.

Note Care should be taken when modifying the client passed to an extension function. In particular, registering callbacks with this client could disrupt the input, output, or event handling of the debugger. It is recommended that a new client be created to register callbacks.

Writing DbgEng Extension Code

10/25/2019 • 2 minutes to read • Edit Online

DbgEng extension commands can include any standard C++ code. They can also include the C++ interfaces that appear in the dbgeng.h header file, in addition to the C functions that appear in the wdbgexts.h header file.

If you intend to use functions from wdbgexts.h, you need to define KDEXT_64BIT before wdbgexts.h is included. For example:

```
#define KDEXT_64BIT
#include wdbgexts.h
#include dbgeng.h
```

For a full list of interfaces in dbgeng.h that can be used in an extension command, see Debugger Engine Reference.

For a full list of functions in wdbgexts.h that can be used in an extension command, see WdbgExts Functions. A number of these functions appear in 32-bit versions and 64-bit versions. Typically, the 64-bit versions end in "64" and the 32-bit versions have no numerical ending -- for example, **ReadIoSpace64** and **ReadIoSpace**. When calling a wdbgexts.h function from a DbgEng extension, you should always use the function name ending in "64". This is because the debugger engine always uses 64-bit pointers internally, regardless of the target platform.

If you include wdbgexts.h in your DbgEng extension, you should call **GetWindbgExtensionApis64** during the initialization of your extension DLL (see *DebugExtensionInitialize*).

Note You must not attempt to call any DbgHelp or ImageHlp routines from any debugger extension. Calling these routines is not supported and may cause a variety of problems.

Building DbgEng Extensions

4/2/2019 • 2 minutes to read • Edit Online

All debugger extensions should be compiled and built by using Visual Studio. The Build utility is no longer used with debugger extensions.

For documentation on building projects in Visual Studio refer to Visual Studio projects - C++.

To build an extension, use the following procedure:

To build a debugger extension

1. Open the **dbgsdk.sln** sample project in Visual Studio.
2. Check the include and lib file project settings. If *%debuggers%* represents the root of your Debugging Tools for Windows installation, they should be set as follows:

```
Include Path
%debuggers%\sdk\inc
Library Path
%debuggers%\sdk\lib
```

If you have moved these headers and libraries to a different location, specify that location instead.

3. Select **Build** and then **Build Solution** from the menu in Visual Studio.

EngExtCpp Extensions

10/23/2019 • 2 minutes to read • Edit Online

This section includes:

EngExtCpp Extension Design Guide

EngExtCpp Extension Reference

EngExtCpp Extension Design Guide

12/5/2018 • 2 minutes to read • Edit Online

This section includes:

EngExtCpp Extension Libraries

Client Objects and the Engine

Writing EngExtCpp Extensions

Building ExtEngCpp Extensions

Parsing Extension Arguments

Typed Data

EngExtCpp Extension Libraries

10/23/2019 • 2 minutes to read • Edit Online

An EngExtCpp extension library is a DLL that uses the EngExtCpp extension framework found in EngExtCpp.h. When this library is loaded by the debugger engine, its methods and functions can provide extra functionality or automation of tasks while performing user-mode or kernel-mode debugging on Microsoft Windows.

The EngExtCpp extension framework is built on top of the DbgEng extension framework. It offers the same debugger engine API for interaction with the debugger engine. but it also provides additional features to make common tasks simpler.

If you performed a full install of Debugging Tools for Windows, a sample EngExtCpp extension called "extcpp" can be found in the sdk\samples\extcpp subdirectory of the installation directory.

EXT_CLASS and ExtExtension

At the core of an EngExtCpp extension library is a single instance of the **EXT_CLASS** class. An EngExtCpp extension library will provide the implementation of this class, which contains all the extension commands and methods for formatting structures that are exported by the library.

EXT_CLASS is a subclass of **ExtExtension**. The single instance of this class is created using the **EXT_DECLARE_GLOBALS** macro which must appear exactly once in the source files for the extension library.

When the extension library is loaded, the **Initialize** method of the class is called by the engine, and the **Uninitialize** method is called before unloading the class. Additionally, the methods **OnSessionActive**, **OnSessionInactive**, **OnSessionAccessible**, and **OnSessionInaccessible** are called by the engine to notify the extension library of the state of the debugging session.

Extension Commands

The **EXT_CLASS** class can contain a number of methods that are used to execute extension commands. Each extension command is declared in the EXT_CLASS class by using the **EXT_COMMAND_METHOD** macro. The implementation of a command is defined by using the **EXT_COMMAND** macro.

Known Structures

The **EXT_CLASS** class can contain a number of methods that use the *ExtKnownStructMethod* prototype. The methods can be used by the engine to format instances of certain structure types for output.

Provided Values

The **EXT_CLASS** class can contain a number of methods that use the **ExtProvideValueMethod** prototype. The methods can be used by the engine to evaluate some pseudo-registers provided by the extension.

Client Objects and the Engine

10/23/2019 • 2 minutes to read • Edit Online

An EngExtCpp extension interacts with the debugger engine through a client object. Interface pointers to the client object are available to the extension through members of the **ExtExtension** base class. The following members provide access to the first version of the engine API interfaces.

ENGINE API INTERFACE	EXTEXTENSION MEMBER
IDebugAdvanced	**m_Advanced**
IDebugClient	**m_Client**
IDebugControl	**m_Control**
IDebugDataSpaces	**m_Data**
IDebugRegisters	**m_Registers**
IDebugSymbols	**m_Symbols**
IDebugSystemObjects	**m_System**

The following members provide access to later versions of the engine API interfaces. These interfaces may not be available in all versions of the debugger engine. If they are not available, any attempt to use them will result in a exception being thrown.

ENGINE API INTERFACE	EXTEXTENSION MEMBER
IDebugAdvanced2	**m_Advanced2**
IDebugAdvanced3	**m_Advanced3**
IDebugClient2	**m_Client2**
IDebugClient3	**m_Client3**
IDebugClient4	**m_Client4**
IDebugClient5	**m_Client5**

ENGINE API INTERFACE	EXTEXTENSION MEMBER
IDebugControl2	**m_Control2**
IDebugControl3	**m_Control3**
IDebugControl4	**m_Control4**
IDebugData2	**m_Data2**
IDebugData3	**m_Data3**
IDebugData4	**m_Data4**
IDebugRegisters2	**m_Registers2**
IDebugSymbols2	**m_Symbols2**
IDebugSymbols3	**m_Symbols3**
IDebugSystemObjects2	**m_System2**
IDebugSystemObjects3	**m_System3**
IDebugSystemObjects4	**m_System4**

The members in these tables are initialized each time the extension library is used to execute an extension command or format a structure for output. Once a task is completed, these members are uninitialized. Consequently, extensions should not cache the values of these members and should use the **ExtExtension** members directly.

An extension library can also create its own client objects using the method **IDebugClient::CreateClient** or the functions **DebugCreate** or **DebugConnect**.

For an overview of client objects, see Client Objects.

Writing EngExtCpp Extensions

6/25/2019 • 2 minutes to read • Edit Online

The EngExtCpp extension library can include any standard C++ code. It can also include the C++ interfaces that appear in the engextcpp.h and dbgeng.h header files, in addition to the C functions that appear in the wdbgexts.h header file. Both dbgeng.h and wdbgexts.h are included from engextcpp.h.

For a full list of interfaces in dbgeng.h that can be used in an extension command, see Debugger Engine Reference.

For a full list of functions in wdbgexts.h that can be used in an extension command, see WdbgExts Functions. A number of these functions appear in 32-bit versions and 64-bit versions. Typically, the 64-bit versions end in "64" and the 32-bit versions have no numerical ending -- for example, **ReadIoSpace64** and **ReadIoSpace**. When calling a wdbgexts.h function from a DbgEng extension, you should always use the function name ending in "64". This is because the debugger engine always uses 64-bit pointers internally, regardless of the target platform. When including wdbgexts.h, engextcpp.h selects the 64-bit version of the API. The **ExtensionApis** global variable used by the WDbgExts API is automatically initialized on entry to a EngExtCpp method and cleared on exit.

When an EngExtCpp extension is used with remote DbgEng interfaces, the WDbgExts interfaces will not be available and the **ExtensionApis** structure can be zeroed. If an EngExtCpp extension is expected to function in such an environment, it should avoid using the WDbgExts API.

Note You must not attempt to call any DbgHelp or ImageHlp routines from any debugger extension. Calling these routines is not supported and may cause a variety of problems.

Building EngExtCpp Extensions

10/23/2019 • 2 minutes to read • Edit Online

The EngExtCpp extension libraries are built almost the same way as the DbgEng extension libraries. For more information, see Building DbgEng Extensions.

The EngExtCpp implementation code (engextcpp.cpp) is used instead of linking with a static library.

Because the EngExtCpp extension framework is built on top of the DbgEng extension framework, an EngExtCpp extension DLL should export the same functions as a DbgEng extension DLL.

Each extension should be exported. When you use the **EXT_COMMAND** macro to define an extension function, this macro also creates a C function with the same name as the extension. This function should be exported from the DLL.

The following functions are provided by engextcpp should be exported from the EngExtCpp DLL.

- **DebugExtensionInitialize** -- so that the **Initialize** method can be called to initialize the library.
- **DebugExtensionUnitialize** -- so that the **Uninitialize** method can be called to uninitialize the library.
- **KnownStructOutputEx** -- so that the engine can call the *ExtKnownStructMethod* methods to format known structures for output.
- **DebugExtensionNotify** -- so that the engine can call the **OnSessionActive**, **OnSessionInactive**, **OnSessionAccessible**, and **OnSessionInaccessible** methods to notify the extension library of changes to the debugging session's state.
- **help** -- so that the EngExtCpp extension framework can automatically provide a **!help** extension.

These functions can be exported even if the functionality they provide is not needed. Moreover, if they are not exported, the functionality they provide will be lost.

DebugExtensionInitialize must be exported in order for the debugger engine to recognize the DLL as a valid DbgEng extension DLL.

Parsing Extension Arguments

10/23/2019 • 5 minutes to read • Edit Online

The EngExtCpp extension framework provides methods to aid in parsing the command-line arguments passed to an extension. To take advantage of these methods, the extension must first declare the format of the command-line arguments in the **EXT_COMMAND** macro.

To bypass the command-line argument parsing done by the framework and let the extension itself parse the arguments, set the command-line description to `"{{custom}}"` and use the method **GetRawArgStr** to get the command-line arguments for parsing.

Command-line description strings will automatically be wrapped when printed, to fit the column width of the display. However, newline characters can be embedded in the description strings - using '`\n`' - to start new lines.

The command-line description can be **NULL** or the empty string. If either occurs, it indicates that the extension command does not take any arguments.

Command-Line Description

The description of the command-line arguments is a sequence that contains two types of components: directives and arguments. The description can optionally contain one of each directive and can contain up to 64 arguments.

Directives

Directives specify how the arguments are parsed. They are enclosed by double braces (`'{{'` and `'}}'`). Each directive can optionally appear zero or one times in the string that describes the arguments.

The following directives are available:

`custom`
Turns off the parsing done by the extension framework and lets the extension perform its own parsing.

`l:str`
Overrides the default long description of the command-line arguments. The extension framework will use *str* for the full description of all the arguments.

`opt:str`
Overrides the default prefix characters for named commands. The default value is `"/-"`, allowing '`/`' or '`-`' to be used as the prefix that identifies named arguments.

`s:str`
Overrides the default short description of the command-line arguments. The extension framework will use *str* for the short description of all the arguments.

Here are some examples of directives. The following string is used by an extension command that parses its own arguments. It also provides short and long descriptions for use with the automatic **!help** extension command:

```
{{custom}}{{s:<arg1> <arg2>}}{{l:arg1 - Argument 1\narg2 - Argument 2}}
```

The following string changes the argument option prefix characters to '`/`' or '`-`'. With this directive, the arguments will be specified using '`+arg`' and '`:arg`' instead of '`/arg`' and '`-arg`':

```
{{opt:+:}}
```

Arguments

Arguments can be of two types: named and unnamed. Unnamed arguments are read positionally. Both types of argument also have a display name, used by the help command.

Argument descriptions are enclosed by single braces (`'{'` and `'}'`).

Each argument description has the following syntax:

```
{[optname];[type[,flags]];[argname];[argdesc]}
```

where:

optname
The name of the argument. This is the name used in commands and in methods that fetch arguments by name. This name is optional. If it is present, the argument becomes a "named argument"; it can appear anywhere on the command-line and is referenced by name. If it is not present, the argument becomes an "unnamed argument"; its position on the command-line is important and it is referenced by its position relative to the other unnamed arguments.

type
The type of the argument. This affects how the argument is parsed and how it is retrieved. The *type* parameter can have one of the following values:

b
Boolean type. The argument is either present or not present. Named Boolean arguments can be retrieved using **HasArg**.

e[d][s][bits]
Expression type. The argument has a numeric value. Named expression arguments can be retrieved using **GetArgU64** and unnamed expression arguments can be retrieved using **GetUnnamedArgU64**.

d
The expression is limited to the next space character in the argument string. If this is not present, the expression evaluator will consume characters from the command line until it determines that it reached the end of the expression.

s
The value of the expression is signed. Otherwise, the value of the expression is unsigned.

bits
The number of bits in the value of the argument. The maximum value for *bits* is 64.

s
String type. The string is limited to the next space character. Named string arguments can be retrieved using **GetArgStr** and unnamed string arguments can be retrieved using **GetUnnamedArgStr**.

x
String type. The argument is the rest of the command line. The argument is retrieved using **GetArgStr** or **GetUnnamedArgStr**, as with the s string type.

flags
The argument flags. These determine how the argument will be treated by the parser. The *flags* parameter can have one of the following values:

d=expr
The default value of the argument. If the argument is not present on the command line, then the argument is set to *expr*. The default value is a string that is parsed according to the type of the argument.

ds
The default value will not be displayed in the argument description provided by the help.

o
The argument is optional. This is the default for named arguments.

r
The argument is required. This is the default for unnamed arguments.

argname
The display name of the argument. This is the name used by the automatic **!help** extension command and by the automatic **/?** or **-?** command-line arguments. Used when printing a summary of the command-line options.

argdesc
A description of the argument. This is the description printed by the automatic **!help** extension and by the automatic "**/?**" or "**-?**" command-line arguments.

Here are some examples of argument descriptions. The following expression defines a command which takes a single optional expression argument. The argument must fit in 32 bits. If the argument isn't present on the command line, the default value of 0x100 will be used.

```
{;e32,o,d=0x100;flags;Flags to control command}
```

The following expression defines a command with an optional Boolean "**/v**" argument and a required unnamed string argument.

```
{v;b;;Verbose mode}{;s;name;Name of object}
```

The following expression defines a command that has an optional named expression argument **/oname** *expr* and an optional named string argument **/eol** *str*. If **/eol** is present, its value will be set to the remainder of the command line and no further arguments will be parsed.

```
{oname;e;expr;Address of object}{eol;x;str;Commands to use}
```

Command Line

The following is a list of some ways that arguments are parsed on the command line:

- The values of named expression and string arguments follow the name on the command line. For example, **/name** *expr* or **/name** *str*.
- For named Boolean arguments, the value is true if the name appears on the command line; false otherwise.
- Multiple single-character-named Boolean options can be grouped together on the command line. For example, "/a /b /c" can be written using the shorthand notation "/abc" (unless there is already an argument named "abc").
- If the command line contains the named argument "?" - for example, "/?" and "-?" - the argument parsing ends, and the help text for the extension is displayed.

Parsing Internals

Several methods are used by the argument parser to set arguments.

The method **SetUnnamedArg** will change the value of an unnamed argument. And, for convenience, the methods **SetUnnamedArgStr** and **SetUnnamedArgU64** will set unnamed string and expression arguments respectively.

Similar methods exist for named arguments. **SetArg** is used to change the value of any named argument and **SetArgStr** and **SetArgU64** are used for named string and expression arguments respectively.

Typed Data

10/23/2019 • 2 minutes to read • Edit Online

The EngExtCpp extension framework provides a few classes to help manipulate the target's memory. The **ExtRemoteData** class describes a small piece of the target's memory. If the type of this memory is known, it is referred to as *typed data* and is described by **ExtRemoteTyped** objects.

Windows lists can be iterated over by using **ExtRemoteList** and, if the type of the objects in the list is known, **ExtRemoteTypedList**.

Note Like the client objects in **ExtExtension**, instances of these classes are only valid while the extension library is used to execute an extension command or format a structure for output. In particular, they should not be cached. For more information about when client objects are valid, see Client Objects and the Engine, .

Remote Data

Remote data should be handled using the class **ExtRemoteData**. This class is a wrapper around a small section of a target's memory. **ExtRemoteData** automatically retrieves the memory and wraps other common requests with throwing methods.

Remote Typed Data

If the type of the remote data is known, it should be handled using the **ExtRemoteTyped** class. This class is an enhanced remote data object that understands data typed with type information from symbols. It is initialized to a particular object by symbol or cast, after which it can be used like an object of the given type.

Remote Lists

To handle remote lists, use the **ExtRemoteList** class. This class can be used for either a singly linked or doubly linked list. If the list is doubly linked, it is assumed that the previous pointer immediately follows the next pointer. The class contains methods that can iterate over the list and retrieve nodes both forward and backward. **ExtRemoteList** can be used with either null-terminated or circular lists also.

Remote Typed Lists

To handle remote lists when the type of the nodes in the list is known, use the **ExtRemoteTypedList** class. This is an enhanced version of **ExtRemoteList**. In addition to the basic functionality of **ExtRemoteList**, **ExtRemoteTypedList** automatically determines link offsets from type information.

Writing WdbgExts Extensions

10/23/2019 • 2 minutes to read • Edit Online

WdbgExts extensions are the original kind of debugger extensions. They are less powerful than DbgEng extensions, but they still offer a wide range of functionality when performing user-mode or kernel-mode debugging on Microsoft Windows.

If you performed a full install of Debugging Tools for Windows, a sample WdbgExts extension called "simplext" can be found in the sdk\samples\simplext subdirectory of the installation directory.

This section includes:

WdbgExts Extension Design Guide

WdbgExts Extension Reference

WdbgExts Extension Design Guide

12/5/2018 • 2 minutes to read • Edit Online

This section includes:

WdbgExts Extension API Overview

32-Bit Pointers and 64-Bit Pointers

Using WdbgExts Extension Callbacks

Using the DECLARE_API Macro

Writing WdbgExts Extension Code

Building WdbgExts Extensions

WdbgExts Extension API Overview

12/5/2018 • 2 minutes to read • Edit Online

Each WdbgExts extension DLL exports one or more functions that are used to implement *extension commands*. These functions are named according to the standard C convention, except that upper-case letters are not permitted.

The function name and the extension command name are identical, except that the extension command begins with an exclamation point (**!**). For example, when you load Myextension.dll into the debugger and then type **!stack** into the Debugger Command window, the debugger looks for an exported function named **stack** in Myextension.dll.

If Myextension.dll is not already loaded, or if there are other extension commands with the same name in other extension DLLs, you can type **!myextension.stack** into the Debugger Command window to indicate the extension DLL and the extension command in that DLL.

Each WdbgExts extension DLL also exports a number of *callback functions*. These functions are called by the debugger when the DLL is loaded and when extension commands are used.

The debugger engine will place a **try / except** block around a call to an extension DLL. This protects the engine from some types of bugs in the extension code. However, since the extension calls are executed in the same thread as the engine, they can still cause the engine to crash.

32-Bit Pointers and 64-Bit Pointers

12/5/2018 • 2 minutes to read • Edit Online

The WdbgExts.h header file supports both 32-bit and 64-bit pointers. To use 64-bit pointers, simply include the following two lines in your code, in the following order:

```
#define KDEXT_64BIT
#include wdbgexts.h
```

It is recommended that you always use 64-bit pointers in your code. This allows your extension to work on any platform, because the debugger will automatically cast 64-bit pointers to 32 bits when the target is 32-bit.

If you intend to use your extension only on 32-bit platforms, you can write a 32-bit extension instead. In that case, you only need to include the following line in your code:

```
#include wdbgexts.h
```

For additional information on working with the 64 bit pointers see Using the DECLARE_API Macro and Writing WdbgExts Extension Code. In addition, examine the sample code that is included as part of the WDK.

Using WdbgExts Extension Callbacks

10/23/2019 • 2 minutes to read • Edit Online

When you write a WdbgExts extension DLL, you can export certain functions:

- You must export a function named *WinDbgExtensionDllInit*. When the debugger loads your extension DLL, it first calls *WinDbgExtensionDllInit* and passes it three arguments.
  - A pointer to a **WINDBG_EXTENSION_APIS64** structure, which contains pointers to functions that are implemented by the debugger and declared in Wdbgexts.h. You must copy the entire structure to a global variable that you create in your DLL.
  - A major version number. You must copy the major version number to a global variable that you create in your DLL.
  - A minor version number. You must copy the minor version number to a global variable that you create in your DLL.

  For example, you could create global variables named ExtensionApis, SavedMajorVersion, and SavedMinorVersion as shown in the following example.

```
WINDBG_EXTENSION_APIS64 ExtensionApis;
USHORT SavedMajorVersion;
USHORT SavedMinorVersion;

VOID WinDbgExtensionDllInit(PWINDBG_EXTENSION_APIS64 lpExtensionApis,
    USHORT MajorVersion, USHORT MinorVersion)
{
   ExtensionApis = *lpExtensionApis;
   SavedMajorVersion = MajorVersion;
   SavedMinorVersion = MinorVersion;
    ...
}
```

- You must export a function called *ExtensionApiVersion*. The debugger calls this function and expects back a pointer to an **EXT_API_VERSION** structure that contains the version number of the extension DLL. The debugger uses this version number when executing commands like **.chain** and **version** that display the extension version number.
- You can optionally export a function called *CheckVersion*. The debugger calls this routine every time you use an extension command. You can use this to print out version mismatch warnings when your DLL is of a slightly different version than the debugger, but not different enough to prevent it from running.

Using the DECLARE_API Macro

12/5/2018 • 2 minutes to read • Edit Online

Each extension command in a WdbgExts extension DLL is declared using the DECLARE_API macro. This macro is defined in wdbgexts.h.

The basic format of the code for an extension command is:

```
DECLARE_API( myextension )
{
    code for myextension
}
```

The DECLARE_API macro sets up a standard interface for extension commands. For example, if the user passed any arguments to the extension command, the entire argument string will be stored as a string, and a pointer to this string (PCSTR) will be passed to the extension function as **args**.

If you are using 64-bit pointers, the DECLARE_API macro is defined as follows:

```
#define DECLARE_API(s)                          \
    CPPMOD VOID                                 \
    s(                                          \
        HANDLE                hCurrentProcess,  \
        HANDLE                hCurrentThread,   \
        ULONG64               dwCurrentPc,      \
        ULONG                 dwProcessor,      \
        PCSTR                 args              \
     )
```

If you are using 32-bit pointers, DECLARE_API remains the same, except that **dwCurrentPc** will be of the type ULONG instead of ULONG64. However, 64-bit pointers are recommended for any extension that you are writing. See 32-Bit Pointers and 64-Bit Pointers for details.

Writing WdbgExts Extension Code

6/25/2019 • 2 minutes to read • Edit Online

WdbgExts extension commands can call any standard C function, as well as the debugger-related functions that appear in the WdbgExts.h header file.

The WdbgExts functions are intended for use in debugger extension commands only. They are useful for controlling and inspecting the computer or application that is being debugged. The WdbgExts.h header file should be included by any code that is calling these WdbgExts functions.

A number of these functions have 32-bit versions as well as 64-bit versions. Typically, the names of the 64-bit WdbgExts functions end in "64," for example **ReadIoSpace64**. The 32-bit versions have no numerical ending, for example, **ReadIoSpace**. If you are using 64-bit pointers, you should use the function name ending in "64"; if you are using 32-bit pointers, you should use the "undecorated" function name. 64-bit pointers are recommended for any extension that you are writing. See 32-Bit Pointers and 64-Bit Pointers for details.

WdbgExts extensions cannot use the C++ interfaces that appear in the DbgEng.h header file. If you wish to use these interfaces, you should write a DbgEng extension or an EngExtCpp extension instead. Both DbgEng extensions and EngExtCpp extensions can use all the interfaces in DbgEng.h as well as those in WdbgExts.h. For details, see Writing DbgEng Extensions and Writing EngExtCpp Extensions.

Note You must not attempt to call any DbgHelp or ImageHlp routines from a debugger extension. This is not supported and may cause a variety of problems.

The following topics give an overview of various categories of WdbgExts functions:

WdbgExts Input and Output

WdbgExts Memory Access

WdbgExts Threads and Processes

WdbgExts Symbols

WdbgExts Target Information

For a full list of these functions, see WdbgExts Functions.

WdbgExts Input and Output

10/23/2019 • 2 minutes to read • Edit Online

This topic provides a brief overview of how input and output can be performed using the WdbgExts API. For an overview of the input and output streams in the debugger engine, see Input and Output in the Debugger Engine Overview section of this documentation.

The function **dprintf** works in a way similar to the standard C **printf** function and prints a formatted string to the Debugger Command window or, more precisely, the formatted string is sent to the output callbacks. To read a line of input from the debugger engine, use the function **GetInputLine**.

To check for a user-initiated interrupt, use **CheckControlC**. This function should be used in loops to determine if the user has attempted to cancel execution of an extension.

For a more powerful input and output API, see Using Input and Output in the Using the Debugger Engine API section of this documentation.

WdbgExts Memory Access

10/23/2019 • 2 minutes to read • Edit Online

This topic provides a brief overview of how memory access can be performed using the WdbgExts API. For an overview of memory access in the debugger engine, see Memory in the Debugger Engine Overview section of this documentation.

Virtual Memory

The virtual memory of the target can be read by using the **ReadMemory** function and written using the **WriteMemory** function. Pointers in the target's memory can be read and written by using the **ReadPointer**, **ReadPtr**, and **WritePointer** functions.

To search the virtual memory for a pattern of bytes, use the **SearchMemory** function.

The **TranslateVirtualToPhysical** function can be used to convert a virtual memory address to a physical memory address.

The **Disasm** function can be used to disassemble a single assembly instruction on the target.

To check the low 4 GB of memory for corruption when using physical address extension (PAE), use the **Ioctl** operation **IG_LOWMEM_CHECK**.

Physical Memory

Physical memory can only be directly accessed in kernel-mode debugging.

The physical memory on the target can be read by using the **ReadPhysical** and **ReadPhysicalWithFlags** functions, and written by using the **WritePhysical** and **WritePhysicalWithFlags** functions.

To search the physical memory for pointers to locations within a specified range, use the **Ioctl** operation **IG_POINTER_SEARCH_PHYSICAL**.

Other Data Spaces

In kernel-mode debugging, it is possible to read and write data to a variety of data spaces in addition to the main memory. The following data spaces can be accessed:

Control-Space Memory
The functions **ReadControlSpace**, **ReadControlSpace64**, **ReadTypedControlSpace32**, and **ReadTypedControlSpace64** will read data from a control space. The **WriteControlSpace** function will write data to a control space.

I/O Memory
The functions **ReadIoSpace**, **ReadIoSpace64**, **ReadIoSpace64**, **ReadIoSpaceEx64** will read data from system I/O memory and bus I/O memory. The functions **WriteIoSpace**, **WriteIoSpace64**, **WriteIoSpaceEx**, and **WriteIoSpaceEx64** will write data to system I/O memory and bus I/O memory.

Model Specific Register (MSR)
The functions **ReadMsr** and **WriteMsr** read and write MSRs.

System Bus
The **Ioctl** operations **IG_GET_BUS_DATA** and **IG_SET_BUS_DATA** read and write system bus data.

Additional Information

For a more powerful memory access API, see Memory Access in the Using the Debugger Engine API section of this documentation.

WdbgExts Threads and Processes

10/23/2019 • 2 minutes to read • Edit Online

This topic provides a brief overview of how threads and processes can be manipulated using the WdbgExts API. For an overview of threads and processes in the debugger engine, see Threads and Processes in the Debugger Engine Overview section of this documentation.

Threads

To get the address of the thread environment block (TEB) that describes the current thread, use the method **GetTebAddress**. In kernel-mode debugging, the KTHREAD structure is also available to describe a thread. This structure is returned by **GetCurrentThreadAddr** (in user-mode debugging, **GetCurrentThreadAddr** returns the address of the TEB).

The thread context is the state preserved by Windows when switching threads; it is represented by the CONTEXT structure. This structure varies with the operating system and platform and care should be taken when using the CONTEXT structure. The thread context is returned by the **GetContext** function and can be set using the **SetContext** function.

To examine the stack trace for the current thread, use the **StackTrace** function. To temporarily change the thread used for examining the stack trace, use the **SetThreadForOperation** or **SetThreadForOperation64** functions. See Examining the Stack Trace in the Using the Debugger Engine API section of this documentation for additional methods for examining the stack.

To get information about an operating system thread in the target, use the **Ioctl** operation **IG_GET_THREAD_OS_INFO**.

Processes

To get the address of the process environment block (PEB) that describes the current process use the method **GetPebAddress**. In kernel-mode debugging, the KPROCESS structure is also available to describe a process. This structure is returned by **GetCurrentProcessAddr** (in user-mode debugging, **GetCurrentProcessAddr** returns the address of the PEB).

The method **GetCurrentProcessHandle** returns the system handle for the current process.

Additional Information

For a more powerful thread manipulation and process manipulation API, see Controlling Threads and Processes in the Using the Debugger Engine API section of this documentation.

WdbgExts Symbols

10/23/2019 • 2 minutes to read • Edit Online

This topic provides a brief overview of how symbols can be manipulated using the WdbgExts API. For an overview of using symbols in the debugger engine, see Symbols in the Debugger Engine Overview section of this documentation.

To evaluate a MASM or C++ expression, use the functions **GetExpression** or **GetExpressionEx**.

To read the value of a member in a structure, use the **GetFieldData** function or, if, the member contains a primitive value, **GetFieldValue** can be used. To determine the size of an instance of a symbol in the target's memory, use the **GetTypeSize** function.

To locate the offset of a member in a structure, use the **GetFieldOffset** function.

To read multiple members in a structure, first use the **InitTypeRead** function to initialize the structure. Then, you can use the **ReadField** function to read the members with size less than or equal to 8 bytes one at a time. For structure addresses in physical memory, use the **InitTypeReadPhysical** function instead of **InitTypeRead**.

There are two functions that you can use for iterating over linked lists. For doubly-linked lists that use the LIST_ENTRY32 or LIST_ENTRY64 structures, the function **ReadListEntry** can be used to find the next and previous entries. The function **ListType** will iterate over all the entries in a linked list and call a callback function for each entry.

To locate a symbol near a specified address in the target's memory, use the **GetSymbol** function.

To delete all the symbol information from the debugger engine's cache, use the **ReloadSymbols** function. To read or change the symbol path, which is used to search for symbol files, use the **GetSetSympath** function.

Almost all symbol operations provided by the debugger engine can be executed using the **Ioctl** operation **IG_DUMP_SYMBOL_INFO**. However, while being a very flexible function, it is advanced and we recommend that you use the above simpler functions where applicable.

Additional Information

For a more powerful symbols API, see Using Symbols in the Using the Debugger Engine API section of this documentation.

WdbgExts Target Information

10/23/2019 • 2 minutes to read • Edit Online

To determine if the target uses 32-bit or 64-bit pointers for memory addresses, use the function **IsPtr64**.

For information about the target's operating system, use the **Ioctl** operation **IG_GET_KERNEL_VERSION**. To get the total number of processors on the target and find out which one is the current processor, use the function **GetKdContext**.

The **GetDebuggerData** function returns a KDDEBUGGER_DATA64 or KDDEBUGGER_DATA32 structure that contains information about the target that the debugger engine has queried or determined during the current session. This information includes certain key target locations and specific status values.

The debugger caches some information obtained from the target. The function **GetDebuggerCacheSize** will return the size of this cache.

Additional Information

For a more powerful target API, see Target Information in the Using the Debugger Engine API section of this documentation.

Building WdbgExts Extensions

12/5/2018 • 2 minutes to read • Edit Online

All debugger extensions should be compiled and built with the Build utility. The Build utility is included in the Windows Driver Kit (WDK) and in earlier versions of the Windows DDK.

Note the following points:

- The WDK has several different build environment windows. Each of these has a corresponding shortcut placed in the **Start** menu when the WDK is installed. To build a debugger extension, you must use the latest Windows build environment, regardless of what platform you will be running the extension on.
- The Build utility is usually not able to compile code that is located in a directory path that contains spaces. Your extension code should be located in a directory whose full path contains no spaces. (In particular, this means that if you install Debugging Tools for Windows to the default location -- Program Files\Debugging Tools for Windows -- you will not be able to build the sample extensions.)

To build a debugger extension

1. Open the window for the latest Windows build environment. (You can choose either the "free" version or the "checked" version -- they will give identical results unless you have put **#ifdef DBG** statements in your code.)
2. Set the variable _NT_TARGET_VERSION to indicate the oldest version of Windows on which you want to run the extension. _NT_TARGET_VERSION can be set to the following values.

VALUE	VERSIONS OF WINDOWS
_NT_TARGET_VERSION_WIN2K	Windows 2000 and later.
_NT_TARGET_VERSION_WINXP	Windows XP and later.
_NT_TARGET_VERSION_WS03	Windows Server 2003 and later.
_NT_TARGET_VERSION_LONGHORN	Windows Vista and later.

If _NT_TARGET_VERSION is not set, the extension will only run on the version of Windows for which the build window was opened (and later versions). For example, putting the following line in your Sources file will build an extension that will run on Windows: `console _NT_TARGET_VERSION = $(_NT_TARGET_VERSION_WINXP)`

3. Set the DBGSDK_INC_PATH and DBGSDK_LIB_PATH environment variables to specify the paths to the debugger SDK headers and the debugger SDK libraries, respectively. If *%debuggers%* represents the root of your Debugging Tools for Windows installation, these variables should be set as follows:

```
set DBGSDK_INC_PATH=%debuggers%\sdk\inc
set DBGSDK_LIB_PATH=%debuggers%\sdk\lib
```

   If you have moved these headers and libraries to a different location, specify that location instead.

4. Change the current directory to the directory that contains your extension's Dirs file or Sources file.

5. Run the Build utility:

```
build -cZMg
```

For a full explanation of these steps, and for a description of how to create a Dirs file and a Sources file, see the Build utility documentation in the WDK.

Writing Custom Analysis Debugger Extensions

12/5/2018 • 2 minutes to read • Edit Online

In this section

- Writing an Analysis Extension Plug-in to Extend !analyze
- Metadata Files for Analysis Extension Plug-ins
- Failure Analysis Entries

Writing an Analysis Extension Plugin to Extend !analyze

10/23/2019 • 7 minutes to read • Edit Online

You can extend the capabilities of the **!analyze** debugger command by writing an analysis extension plugin. By providing an analysis extension plugin, you can participate in the analysis of a bug check or an exception in a way that is specific to your own component or application.

When you write an analysis extension plugin, you also write a metadata file that describes the situations for which you want your plugin to be called. When **!analyze** runs, it locates, loads, and runs the appropriate analysis extension plugins.

To write an analysis extension plugin and make it available to **!analyze**, follow these steps.

- Create a DLL that exports an **_EFN_Analyze** function.
- Create a metadata file that has the same name as your DLL and an extension of .alz. For example, if your DLL is named MyAnalyzer.dll, your metadata file must be named MyAnalyzer.alz. For information about how to create a metadata file, see Metadata Files for Analysis Extensions. Place the metadata file in the same directory as your DLL.
- In the debugger, use the **.extpath** command to add your directory to the extension file path. For example, if your DLL and metadata file are in the folder named c:\MyAnalyzer, enter the command **.extpath+ c:\MyAnalyzer**.

When the **!analyze** command runs in the debugger, the analysis engine looks in the extension file path for metadata files that have the .alz extension. The analysis engine reads the metadata files to determine which analysis extension plugins should be loaded. For example, suppose the analysis engine is running in response to Bug Check 0xA IRQL_NOT_LESS_OR_EQUAL, and it reads a metadata file named MyAnalyzer.alz that contains the following entries.

```
PluginId       MyPlugin
DebuggeeClass  Kernel
BugCheckCode   0xA
BugCheckCode   0xE2
```

The entry `BugCheckCode 0x0A` specifies that this plugin wants to participate in the analysis of Bug Check 0xA, so the analysis engine loads MyAnalyzer.dll (which must be in the same directory as MyAnalyzer.alz) and calls its **_EFN_Analyze** function.

Note The last line of the metadata file must end with a newline character.

Skeleton Example

Here is a skeleton example that you can use as a starting point.

1. Build a DLL named MyAnalyzer.dll that exports the **_EFN_Analyze** function shown here.

```
#include <windows.h>
#define KDEXT_64BIT
#include <wdbgexts.h>
#include <dbgeng.h>
#include <extsfns.h>

extern "C" __declspec(dllexport) HRESULT _EFN_Analyze(_In_ PDEBUG_CLIENT4 Client,
   _In_ FA_EXTENSION_PLUGIN_PHASE CallPhase, _In_ PDEBUG_FAILURE_ANALYSIS2 pAnalysis)
{
   HRESULT hr = E_FAIL;

   PDEBUG_CONTROL pControl = NULL;
   hr = Client->QueryInterface(__uuidof(IDebugControl), (void**)&pControl);

   if(S_OK == hr && NULL != pControl)
   {
      IDebugFAEntryTags* pTags = NULL;
      pAnalysis->GetDebugFATagControl(&pTags);

      if(NULL != pTags)
      {
         if(FA_PLUGIN_INITILIZATION == CallPhase)
         {
            pControl->Output(DEBUG_OUTPUT_NORMAL, "My analyzer: initialization\n");
         }
         else if(FA_PLUGIN_STACK_ANALYSIS == CallPhase)
         {
            pControl->Output(DEBUG_OUTPUT_NORMAL, "My analyzer: stack analysis\n");
         }
         else if(FA_PLUGIN_PRE_BUCKETING == CallPhase)
         {
            pControl->Output(DEBUG_OUTPUT_NORMAL, "My analyzer: prebucketing\n");
         }
         else if(FA_PLUGIN_POST_BUCKETING == CallPhase)
         {
            pControl->Output(DEBUG_OUTPUT_NORMAL, "My analyzer: post bucketing\n");
            FA_ENTRY_TYPE entryType = pTags->GetType(DEBUG_FLR_BUGCHECK_CODE);
            pControl->Output(DEBUG_OUTPUT_NORMAL, "The data type for the DEBUG_FLR_BUGCHECK_CODE tag is
0x%x.\n\n", entryType);
         }
      }

      pControl->Release();
   }
   return hr;
}
```

2. Create a metadata file named MyAnalyzer.alz that has the following entries.

```
PluginId      MyPlugin
DebuggeeClass Kernel
BugCheckCode  0xE2
```

Note The last line of the metadata file must end with a newline character.

3. Establish a kernel-mode debugging session between a host and target computer.

4. On the host computer, put MyAnalyzer.dll and MyAnalyzer.alz in the folder c:\MyAnalyzer.

5. On the host computer, in the debugger, enter these commands.

.extpath+ c:\MyAnalyzer

.crash

6. The **.crash** command generates Bug Check 0xE2 MANUALLY_INITIATED_CRASH on the target computer, which causes a break in to the debugger on the host computer. The bug check analysis engine (running in the debugger on the host computer) reads MyAnalyzer.alz and sees that MyAnalyzer.dll is able to participate in analyzing bug check 0xE2. So the analysis engine loads MyAnalyzer.dll and calls its **_EFN_Analyze** function.

   Verify that you see output similar to the following in the debugger.

```
*                        Bugcheck Analysis                                    *
*                                                                             *
*******************************************************************************

Use !analyze -v to get detailed debugging information.

BugCheck E2, {0, 0, 0, 0}

My analyzer: initialization
My analyzer: stack analysis
My analyzer: prebucketing
My analyzer: post bucketing
The data type for the DEBUG_FLR_BUGCHECK_CODE tag is 0x1.
```

The preceding debugger output shows that the analysis engine called the **_EFN_Analyze** function four times: once for each phase of the analysis. The analysis engine passes the **_EFN_Analyze** function two interface pointers. *Client* is an **IDebugClient4** interface, and *pAnalysis* is an **IDebugFailureAnalysis2** interface. The code in the preceding skeleton example shows how to obtain two more interface pointers. `Client->QueryInterface` gets an **IDebugControl** interface, and `pAnalysis->GetDebugFATagControl` gets an **IDebugFAEntryTags** interface.

Failure Analysis Entries, Tags, and Data Types

The analysis engine creates a **DebugFailureAnalysis** object to organize the data related to a particular code failure. A **DebugFailureAnalysis** object has a collection of failure analysis entries (FA entries), each of which is represented by an **FA_ENTRY** structure. An analysis extension plugin uses the **IDebugFailureAnalysis2** interface to get access to this collection of FA entries. Each FA entry has a tag that identifies the kind of information that the entry contains. For example, an FA entry might have the tag **DEBUG_FLR_BUGCHECK_CODE**, which tells us that the entry contains a bug check code. Tags are values in the **DEBUG_FLR_PARAM_TYPE** enumeration (defined in extsfns.h), which is also called the **FA_TAG** enumeration.

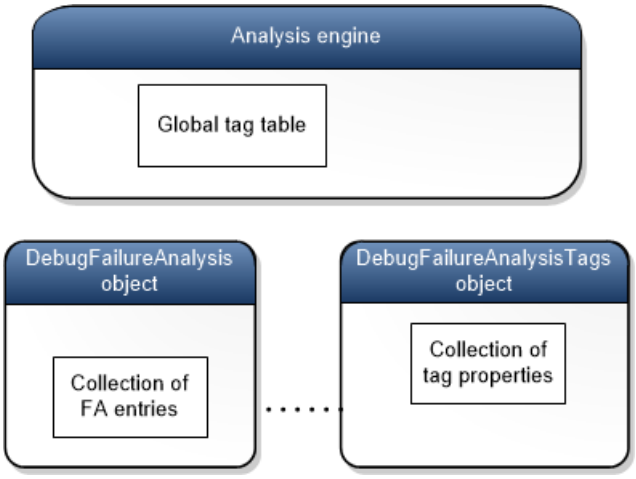
```
typedef enum _DEBUG_FLR_PARAM_TYPE {
    ...
    DEBUG_FLR_BUGCHECK_CODE,
    ...
    DEBUG_FLR_BUILD_VERSION_STRING,
    ...
} DEBUG_FLR_PARAM_TYPE;

typedef DEBUG_FLR_PARAM_TYPE FA_TAG;
```

Most FA entries have an associated data block. The **DataSize** member of the **FA_ENTRY** structure holds the size of the data block. Some FA entries do not have an associated data block; all the information is conveyed by the tag. In those cases, the **DataSize** member has a value of 0.

```
typedef struct _FA_ENTRY
{
    FA_TAG Tag;
    USHORT FullSize;
    USHORT DataSize;
} FA_ENTRY, *PFA_ENTRY;
```

Each tag has a set of properties: for example, name, description, and data type. A **DebugFailureAnalysis** object is associated with a DebugFailureAnalysisTags object, which contains a collection of tag properties. The following diagram illustrates this association.



A **DebugFailureAnalysis** object has a collection of FA entries that belong to a particular analysis session. The associated DebugFailureAnalysisTags object has a collection of tag properties that includes only the tags used by that same analysis session. As the preceding diagram shows, the analysis engine has a global tag table that holds limited information about a large set of tags that are generally available for use by analysis sessions.

Typically most of the tags used by an analysis session are standard tags; that is, the tags are values in the **FA_TAG** enumeration. However, an analysis extension plug-in can create custom tags. An analysis extension plug-in can add an FA entry to a **DebugFailureAnalysis** object and specify a custom tag for the entry. In that case, properties for the custom tag are added to the collection of tag properties in the associated DebugFailureAnalysisTags object.

You can access a DebugFailureAnalysisTags through an IDebugFAEntry tags interface. To get a pointer to an IDebugFAEntry interface, call the **GetDebugFATagControl** method of the **IDebugFailureAnalysis2** interface.

Each tag has a data type property that you can inspect to determine the data type of the data in a failure analysis entry. A data type is represented by a value in the **FA_ENTRY_TYPE** enumeration.

The following line of code gets the data type of the **DEBUG_FLR_BUILD_VERSION_STRING** tag. In this case, the data type is **DEBUG_FA_ENTRY_ANSI_STRING**. In the code, `pAnalysis` is a pointer to an **IDebugFailureAnalysis2** interface.

```
IDebugFAEntryTags* pTags = pAnalysis->GetDebugFATagControl(&pTags);

if(NULL != pTags)
{
   FA_ENTRY_TYPE entryType = pTags->GetType(DEBUG_FLR_BUILD_VERSION_STRING);
}
```

If a failure analysis entry has no data block, the data type of the associated tag is **DEBUG_FA_ENTRY_NO_TYPE**.

Recall that a **DebugFailureAnalysis** object has a collection of FA entries. To inspect all the FA entries in the collection, use the **NextEntry** method. The following example shows how to iterate through the entire collection of FA entries. Assume that *pAnalysis* is a pointer to an **IDebugFailureAnalysis2** interface. Notice that we get the

first entry by passing **NULL** to **NextEntry**.

```
PFA_ENTRY entry = pAnalysis->NextEntry(NULL);

while(NULL != entry)
{
   // Do something with the entry

   entry = pAnalysis->NextEntry(entry);
}
```

A tag can have a name and a description. In the following code, *pAnalysis* is a pointer to an **IDebugFailureAnalysis** interface, *pControl* is a pointer to an **IDebugControl** interface, and `pTags` is a pointer to an IDebugFAEntryTags interface. The code shows how to use the **GetProperties** method to get the name and description of the tag associated with an FA entry.

```
#define MAX_NAME_LENGTH 64
#define MAX_DESCRIPTION_LENGTH 512

CHAR name[MAX_NAME_LENGTH] = {0};
ULONG nameSize = MAX_NAME_LENGTH;
CHAR desc[MAX_DESCRIPTION_LENGTH] = {0};
ULONG descSize = MAX_DESCRIPTION_LENGTH;

PFA_ENTRY pEntry = pAnalysis->NextEntry(NULL);
pTags->GetProperties(pEntry->Tag, name, &nameSize, desc, &descSize, NULL);
pControl->Output(DEBUG_OUTPUT_NORMAL, "The name is %s\n", name);
pControl->Output(DEBUG_OUTPUT_NORMAL, "The description is %s\n", desc);
```

Related topics

Writing Custom Analysis Debugger Extensions

_EFN_Analyze

Metadata Files for Analysis Extension Plug-ins

IDebugFailureAnalysis2

IDebugFAEntryTags

Metadata Files for Analysis Extension Plug-ins

10/23/2019 • 3 minutes to read • Edit Online

When you write an analysis extension plug-in, you also write a metadata file that describes the situations for which you want your plug-in to be called. When the **!analyze** debugger command runs, it uses metadata files to determine which plug-ins to load.

Create a metadata file that has the same name as your analysis extension plug-in and an extension of .alz. For example, if your analysis extension plug-in is named MyAnalyzer.dll, your metadata file must be named MyAnalyzer.alz. Place the metadata file in the same directory as your analysis extension plug-in.

A metadata file for an analysis extension plug-in is an ASCII text file that contains key-value pairs. Keys and values are separated by white space. A key can have any non-whitespace character. Keys are not case sensitive.

After the key and the following white space, the corresponding value begins. A value can have one of the following forms.

- Any set of characters to the end of the line. This form works for values that do not contain any newline characters.

  Important If the last value in the metadata file has a value of this form, the line must end with a newline character.

- Any set of characters between braces { }. The form works for values that contain newline characters.

A line beginning with # is a comment and is ignored. Comments can start only where keys are expected.

You can use the following keys in a metadata file.

KEY	DESCRIPTION
PluginId	String - Identifies the plug-in.
DebuggeeClass	String - Possible values are “Kernel” and "User". Indicates that the plug-in is interested in analyzing only kernel-mode failures or only user-mode failures.
BugCheckCode	32-bit bug check code - Indicates that the plug-in is interested in analyzing this bug check code. A single metadata file can specify multiple bug check codes.
ExceptionCode	32-bit exception code - Indicates that the plug-in is interested in analyzing this exception code. A single metadata file can specify multiple exception codes.
ExecutableName	String - Indicates that the plug-in is interested only in sessions where this is the running executable of the process to be analyzed. A single metadata file can specify multiple executable names.

KEY	DESCRIPTION
ImageName	String - Indicates that the plug-in is only interested only in sessions where the default analysis considers this image (dll, sys, or exe) to be at fault. The plug-in is invoked after analysis has determined which image is at fault. A single metadata file can specify multiple image names.
MaxTagCount	Integer - The maximum number of custom tags that the plug-in needs. Custom tags are tags other than the ones defined in extsfns.h.

Example Metadata Files

The following metadata file describes a plug-in that is interested in analyzing bug check code 0xE2. (Recall that the last line must end with a newline character.)

```
PluginId      MyPlugin
DebuggeeClass Kernel
BugCheckCode  0xE2
```

The following metadata file describes a plug-in that is interested in analyzing bug checks 0x8, 0x9, and 0xA if MyDriver.sys is considered to be the module at fault.

```
PluginId      MyPlugin
DebuggeeClass Kernel
BugCheckCode  0x8
BugCheckCode  0x9
BugCheckCode  0xA
ImageName     MyDriver.sys
```

The following metadata file describes a plug-in that is interested in analyzing exception code 0xC0000005 if MyApp.exe is the running executable of the process being analyzed. Also, the plug-in might create as many as three custom tags.

```
PluginId        MyPlugin
DebuggeeClass   User
ExceptionCode   0xC0000005
ExecutableName  MyApp.exe
```

Debugging Tools for Windows has a sample that you can use to build a debugger extension module named dbgexts.dll. This extension module implements several debugger extension commands, but it can also serve as an analysis extension plug-in; that is, it exports an **_EFN_Analyze** function. Here is a metadata file that describes dbgexts.dll as an analysis extension plug-in.

```
PluginId         PluginSample
DebuggeeClass   User
ExceptionCode   0xc0000005
ExecutableName      cdb.exe
ExecutableName      windbg.exe
#
# Custom tag descriptions
#
TagDesc         0xA0000000  SAMPLE_PLUGIN_DEBUG_TEXT    {Sample debug
help text from plug-in analysis}
#
```

Related topics

Writing an Analysis Extension Plug-in to Extend !analyze

_EFN_Analyze

!analyze

Failure Analysis Entries

10/23/2019 • 2 minutes to read • Edit Online

A **DebugFailureAnalysis** object has a collection of failure analysis entries. For more information, see Failure Analysis Entries, Tags, and Data Types.

A *failure analysis entry* (also called an *FA entry*) is one of the following:

- An **FA_ENTRY** structure
- An **FA_ENTRY** structure followed by a data block

The **DataSize** member of the **FA_ENTRY** structure holds the size, in bytes, of the data block. If there is no data block, **DataSize** is equal to 0. The **Tag** member of an **FA_ENTRY** structure identifies the kind of information that is stored in the FA entry. For example, the tag **DEBUG_FLR_BUGCHECK_CODE** indicates that the data block of the **FA_ENTRY** holds a bug check code.

In some cases, there is no need for a data block; all the information is conveyed by the tag. For example, an **FA_ENTRY** with tag **DEBUG_FLR_KERNEL_VERIFIER_ENABLED** has no data block.

Each tag is associated with one of the data types in the **FA_ENTRY_TYPE** enumeration. For example, the tag **DEBUG_FLR_BUGCHECK_CODE** is associated with the data type **DEBUG_FA_ENTRY_ULONG**. To determine the data type of a tag, call the **GetType** method of the IDebugFAEntryTags interface.

To get or set the data block of an FA entry, use the **IDebugFailureAnalysis2** interface.

Related topics

Writing an Analysis Extension Plug-in to Extend !analyze

FA_ENTRY

Debugger Programming Reference

10/23/2019 • 2 minutes to read • Edit Online

This topic describes the programming reference topics.

This reference section includes:

Debugger Engine Reference

EngExtCpp Extension Reference

WdbgExts Extension Reference

Custom Analysis Debugger Extension Reference

Debug Engine Interfaces

10/23/2019 • 2 minutes to read • Edit Online

This section covers the following interfaces:

- **IDebugAdvanced**
- **IDebugAdvanced2**
- **IDebugAdvanced3**
- **IDebugAdvanced4**
- **IDebugClient**
- **IDebugClient2**
- **IDebugClient3**
- **IDebugClient4**
- **IDebugClient5**
- **IDebugClient6**
- **IDebugClient7**
- **IDebugControl**
- **IDebugControl2**
- **IDebugControl3**
- **IDebugControl4**
- **IDebugControl5**
- **IDebugControl6**
- **IDebugControl7**
- **IDebugDataSpaces**
- **IDebugDataSpaces2**
- **IDebugDataSpaces3**
- **IDebugDataSpaces4**
- **IDebugRegisters**
- **IDebugRegisters2**
- **IDebugSymbols**
- **IDebugSymbols2**
- **IDebugSymbols3**
- **IDebugSymbols4**

- **IDebugSymbols5**
- **IDebugSystemObjects**
- **IDebugSystemObjects2**
- **IDebugSystemObjects3**
- **IDebugSystemObjects4**

Debugger Engine Reference

10/23/2019 • 2 minutes to read • Edit Online

This section includes:

Client Functions

Debug Engine Interfaces

Callback Debug Engine Interfaces

Other Debug Engine Interfaces

Structures and Constants

Certain common methods are supported by all COM interfaces on Microsoft Windows. These methods are not listed individually in this reference section. For details, see Using Client Objects.

DebugBaseEventCallbacks class

10/23/2019 • 2 minutes to read • Edit Online

The DebugBaseEventCallbacks class provides a base implementation of the IDebugEventCallbacks interface.

A program can derive an event callbacks class from DebugBaseEventCallbacks and implement only the methods needed.

Be careful to implement GetInterestMask appropriately.

Requirements

Header

Dbgeng.h (include Dbgeng.h)

See Also

DebugBaseEventCallbacksWide

DebugBaseEventCallbacksWide class

10/23/2019 • 2 minutes to read • Edit Online

The DebugBaseEventCallbacksWide class provides a base implementation of the IDebugEventCallbacksWide interface.

A program can derive an event callbacks class from DebugBaseEventCallbacksWide and implement only the methods needed.

Be careful to implement GetInterestMask appropriately.

Requirements

Header

Dbgeng.h (include Dbgeng.h)

See Also

DebugBaseEventCallbacks

Debug API Constants

12/5/2018 • 2 minutes to read • Edit Online

DBG_ASMOPT_XXX

DBG_ATTACH_XXX

DBG_DUMP_XXX

DBG_DUMP_FIELD_XXX

DBG_ENGOPT_XXX

DBG_EVENT_XXX

DEBUG_FILTER_XXX

DEBUG_FORMAT_XXX

DEBUG_OUTCTL_XXX

DEBUG_OUTPUT_XXX

DEBUG_PROCESS_XXX

DEBUG_STATUS_XXX

DEBUG_SYMBOL_XXX

DEBUG_TYPEOPTS_XXX

DEBUG_ASMOPT_XXX

12/5/2018 • 2 minutes to read • Edit Online

The assembly and disassembly options affect how the debugger engine assembles and disassembles processor instructions for the target.

The options are represented by a bitset with the following bit flags.

CONSTANT	DESCRIPTION
DEBUG_ASMOPT_VERBOSE	When this bit is set, additional information is included in the disassembly. This is equivalent to the **verbose** option in the **.asm** command.
DEBUG_ASMOPT_NO_CODE_BYTES	When this bit is set, the raw bytes for an instruction are not included in the disassembly. This is equivalent to the **no_code_bytes** option in the **.asm** command.
DEBUG_ASMOPT_IGNORE_OUTPUT_WIDTH	When this bit is set, the debugger ignores the width of the output display when formatting instructions during disassembly. This is equivalent to the **ignore_output_width** option in the **.asm** command.

DEB UG_ AS MO PT_ SO URC E_LI NE_ NU MB ER	When this bit is set, each line of the disassembly output is prefixed with the line number of the source code provided by symbol information. This is equivalent to the **source_line** option in the **.asm** command.

Remarks

Additionally, the value DEBUG_ASMOPT_DEFAULT represents the default set of assembly and disassembly options. This means that all the options in the preceding table are turned off.

Requirements

Header	DbgEng.h (include DbgEng.h)

DEBUG_ATTACH_XXX

10/23/2019 • 2 minutes to read • Edit Online

The DEBUG_ATTACH_*XXX* bit-flags described in this topic control how the debugger engine attaches to a user-mode process. For the DEBUG_ATTACH_XXX options used when attaching to a kernel target, see the IDebugClient::AttachKernel method.

The possible values include the following.

CONSTANT	DESCRIPTION
DEBUG_ATTACH_NONINVASIVE	Attach to the target noninvasively. For more information about noninvasive debugging, see Noninvasive Debugging (User Mode). If this flag is set, then the flags DEBUG_ATTACH_EXISTING, DEBUG_ATTACH_INVASIVE_NO_INITIAL_BREAK, and DEBUG_ATTACH_INVASIVE_RESUME_PROCESS must not be set.
DEBUG_ATTACH_EXISTING	Re-attach to an application to which a debugger has already attached (and possibly abandoned). For more information about re-attaching to targets, see .attach (Attach to Process). If this flag is set, then the other DEBUG_ATTACH_*XXX* flags must not be set.
DEBUG_ATTACH_NONINVASIVE_NO_SUSPEND	Do not suspend the target's threads when attaching noninvasively. If this flag is set, then the flag DEBUG_ATTACH_NONINVASIVE must also be set.

DEB UG_ ATT AC H_I NV ASI VE_ NO_ INIT IAL_ BRE AK	Do not request an initial break-in when attaching to the target. If this flag is set, then the flags DEBUG_ATTACH_NONINVASIVE and DEBUG_ATTACH_EXISTING must not be set.
DEB UG_ ATT AC H_I NV ASI VE_ RES UM E_P RO CES S	Resume all of the target's threads when attaching invasively. If this flag is set, then the flags DEBUG_ATTACH_NONINVASIVE and DEBUG_ATTACH_EXISTING must not be set.

Requirements

Header	DbgEng.h (include DbgEng.h)

DBG_DUMP_XXX

10/23/2019 • 2 minutes to read • Edit Online

The DBG_DUMP_*XXX* bit flags are used by the **Options** member of the SYM_DUMP_PARAM structure to control the behavior of the **IG_DUMP_SYMBOL_INFOIoctl** operation.

The following flags can be present.

FLAG	EFFECT
DBG_DUMP_NO_INDENT	Members are not indented in the output.
DBG_DUMP_NO_OFFSET	Offsets are not printed.
DBG_DUMP_VERBOSE	Verbose output.
DBG_DUMP_CALL_FOR_EACH	A callback function is called for each member.
DBG_DUMP_LIST	The symbol is an entry in a linked list and the IG_DUMP_SYMBOL_INFO **Ioctl** operation will iterate over this list. The description of the member that points to the next item in the list is specified by the **linkList** member of the SYM_DUMP_PARAM structure.
DBG_DUMP_NO_PRINT	Nothing is printed (only callback functions are called and data copies are performed).
DBG_DUMP_GET_SIZE_ONLY	The **Ioctl** operation returns the size of the symbol only; it will not print member information or call callback functions.
DBG_DUMP_COMPACT_OUT	Newlines are not printed after each member.
DBG_DUMP_ARRAY	The symbol is an array. The number of elements in the array is specified by the member **listLink->size** of the SYM_DUMP_PARAM structure.
DBG_DUMP_ADDRESS_OF_FIELD	The value of **addr** is actually the address of the member **listLink->fName** of the SYM_DUMP_PARAM structure and not the beginning of the symbol.
DBG_DUMP_ADDRESS_AT_END	The value of **addr** is actually the address at the end of the symbol and not the beginning of the symbol.

FLAG	EFFECT
DBG_DUMP_COPY_TYPE_DATA	The value of the symbol is copied into the member **pBuffer**. This can only be used for primitive types--for example, ULONG or PVOID--it cannot be used with structures.
DBG_DUMP_READ_PHYSICAL	The symbol's value will be read directly from the target's physical memory.
DBG_DUMP_FUNCTION_FORMAT	When formatting a symbol that has a function type, the function format will be used, for example, `function(arg1, arg2, ...)`
DBG_DUMP_BLOCK_RECURSE	Recurse through nested structures; but do not follow pointers.

In addition, the result of the macro DBG_DUMP_RECUR_LEVEL(*Level*) can be added to the bit-set to specify how deep into structures to recurse. *Level* can be a number between 0 and 15.

Requirements

Header	Wdbgexts.h (include Wdbgexts.h, Wdbgexts.h, or Dbgeng.h)

See also

IG_DUMP_SYMBOL_INFO

Ioctl

DBG_DUMP_FIELD_XXX

10/23/2019 • 2 minutes to read • Edit Online

The DBG_DUMP_FIELD_*XXX* bit flags are used by the **fOptions** member of the **FIELD_INFO** structure to control the behavior of the **IG_DUMP_SYMBOL_INFOIoctl** operation.

The following flags can be present.

FLAG	EFFECT
DBG_DUMP_FIELD_CALL_BEFORE_PRINT	The callback function is called before printing the member.
DBG_DUMP_FIELD_NO_CALLBACK_REQ	No callback function is called.
DBG_DUMP_FIELD_RECUR_ON_THIS	Submembers of the member are processed.
DBG_DUMP_FIELD_FULL_NAME	**fName** must match completely, as opposed to just having a matching prefix, for the member to be processed.
DBG_DUMP_FIELD_ARRAY	Print array elements of an array member.
DBG_DUMP_FIELD_COPY_FIELD_DATA	The value of the member is copied into **pBuffer**.
DBG_DUMP_FIELD_RETURN_ADDRESS	During a callback or when **Ioctl** returns, the FIELD_INFO.**address** member contains the address of the symbol's member. If no address is supplied for the type, FIELD_INFO.**address** contains total offset of the member from the beginning of the type.
DBG_DUMP_FIELD_SIZE_IN_BITS	For a bit field, return the offset and size in bits instead of bytes.
DBG_DUMP_FIELD_NO_PRINT	Do not print this member (only callback function are called and data copies are performed).
DBG_DUMP_FIELD_DEFAULT_STRING DBG_DUMP_FIELD_WCHAR_STRING DBG_DUMP_FIELD_MULTI_STRING DBG_DUMP_FIELD_GUID_STRING	If the member is a pointer, it is printed as a string, ANSI string , WCHAR string, MULTI string, or GUID.

In addition, the result of the macro DBG_DUMP_RECUR_LEVEL(*Level*) can be added to the bit-set to specify how deep into structures to recurse. *Level* can be a number between 0 and 15.

Requirements

Header	Wdbgexts.h (include Wdbgexts.h, Wdbgexts.h, or Dbgeng.h)

See also

IG_DUMP_SYMBOL_INFO

Ioctl

FIELD_INFO

DEBUG_ENGOPT_XXX

3/5/2019 • 2 minutes to read • Edit Online

The following global options affect the behavior of the debugger engine.

CONSTANT	DESCRIPTION
DEBUG_ENGOPT_IGNORE_DBGHELP_VERSION	The debugger engine generates a warning instead of an error if the version of the DbgHelp DLL does not match the version of the debugger engine.
DEBUG_ENGOPT_IGNORE_EXTENSION_VERSIONS	Disable version checking for extensions. This suppresses the debugger engine's call to CheckVersion.
DEBUG_ENGOPT_ALLOW_NETWORK_PATHS	Network shares can be used for loading symbols and extensions. This option prevents the engine from disallowing network paths when debugging some system processes and should be used with caution. This option cannot be set if DEBUG_ENGOPT_DISALLOW_NETWORK_PATHS is set.

DEBUG_ENGOPT_DISALLOW_NETWORK_PATHS	Network shares cannot be used for loading symbols and extensions. The engine attempts to set this option when debugging some system processes. This option cannot be set if DEBUG_ENGOPT_ALLOW_NETWORK_PATHS is set.
DEBUG_ENGOPT_NETWORK_PATHS	Bitwise OR of DEBUG_ENGOPT_ALLOW_NETWORK_PATHS and DEBUG_ENGOPT_DISALLOW_NETWORK_PATHS.
DEBUG_ENGOPT_IGNORE_LOADER_EXCEPTIONS	Ignore expected first-chance exceptions that are generated by the loader in certain older versions of Windows.
DEBUG_ENGOPT_INITIAL_BREAK	Break into the debugger at the target's initial event.

DEBUG_ENGOPT_INITIAL_MODULE_BREAK	Break into the debugger when the target loads its first module.
DEBUG_ENGOPT_FINAL_BREAK	Break into the debugger at the target's final event. In a live user-mode target, this is when the process exits. It has no effect in kernel mode.
DEBUG_ENGOPT_NO_EXECUTE_REPEAT	When given an empty command, the debugger engine does not repeat the last command.
DEBUG_ENGOPT_FAIL_INCOMPLETE_INFORMATION	Prevent the debugger from loading modules whose images cannot be mapped. The debugger attempts to load images when debugging minidumps that do not contain images.

DEB UG_ ENG OPT _AL LO W_ REA D_O NLY _BR EAK POI NTS	Allow the debugger engine to manipulate page protections on the target to allow for setting software breakpoints in a read-only section of memory. When setting software breakpoints, the engine transparently alters the target's memory to insert an interrupt instruction.
DEB UG_ ENG OPT _SY NC HR ONI ZE_ BRE AKP OIN TS	In live user-mode debugging, the engine performs extra work when inserting and removing breakpoints to ensure that all threads in the target have a consistent breakpoint state at all times. This option is useful when multiple threads can use the code for which the breakpoint is set. However, it can introduce the possibility of deadlocks.
DEB UG_ ENG OPT _DIS ALL OW _SH ELL_ CO MM AN DS	Disallow executing shell commands through the debugger. After this option has been set, it cannot be unset.
DEB UG_ ENG OPT _KD _QU IET_ MO DE	Turn on quiet mode. For more information, see **sq (Set Quiet Mode)**.

DEBUG_ENGOPT_DISABLE_MANAGED_SUPPORT	Disables debugger engine support for managed code. If support for managed code is already in use, this option has no effect.
DEBUG_ENGOPT_DISABLE_MODULE_SYMBOL_LOAD	The debugger does not load symbols for modules that are loaded while this flag is set.
DEBUG_ENGOPT_DISABLE_EXECUTION_COMMANDS	Prevents any commands that would cause the target to begin executing.
DEBUG_ENGOPT_DISALLOW_IMAGE_FILE_MAPPING	Disallows mapping of image files from disk. For example, this option disallows image mapping for memory content during debugging of minidump files. This option does not affect existing mappings; it affects only subsequent attempts to map image files.

DEB UG_ ENG OPT _PR EFE R_D ML	The debugger runs DML-enhanced versions of commands and operations by default.
DEB UG_ ENG OPT _DIS ABL ESQ M	Disables upload of Software Quality Metrics (SQM) data.

Requirements

Header	DbgEng.h (include DbgEng.h)

DEBUG_EVENT_XXX

12/5/2018 • 2 minutes to read • Edit Online

The following events are generated by the target.

FLAG	IDEBUGEVENTCALLBACKSMETHOD	EVENT DESCRIPTION
DEBUG_EVENT_BREAKPOINT	**IDebugEventCallbacks::Breakpoint**	A breakpoint exception occurred in the target.
DEBUG_EVENT_EXCEPTION	**IDebugEventCallbacks::Exception**	An exception debugging event occurred in the target.
DEBUG_EVENT_CREATE_THREAD	**IDebugEventCallbacks::CreateThread**	A create-thread debugging event occurred in the target.
DEBUG_EVENT_EXIT_THREAD	**IDebugEventCallbacks::ExitThread**	An exit-thread debugging event occurred in the target.
DEBUG_EVENT_CREATE_PROCESS	**IDebugEventCallbacks::CreateProcess**	A create-process debugging event occurred in the target.
DEBUG_EVENT_EXIT_PROCESS	**IDebugEventCallbacks::ExitProcess**	An exit-process debugging event occurred in the target.
DEBUG_EVENT_LOAD_MODULE	**IDebugEventCallbacks::LoadModule**	A module-load debugging event occurred in the target.
DEBUG_EVENT_UNLOAD_MODULE	**IDebugEventCallbacks::UnloadModule**	A module-unload debugging event occurred in the target.
DEBUG_EVENT_SYSTEM_ERROR	**IDebugEventCallbacks::SystemError**	A system error occurred in the target.

The following events are generated by the debugger engine.

FLAG	IDEBUGEVENTCALLBACKSMETHOD	DESCRIPTION
DEBUG_EVENT_SESSION_STATUS	**IDebugEventCallbacks::SessionStatus**	A change has occurred in the session status.

DEBUG_EVENT_CHANGE_DEBUGGEE_STATE	**IDebugEventCallbacks::ChangeDebuggeeState**	The engine has made or detected a change in the target status.
DEBUG_EVENT_CHANGE_ENGINE_STATE	**IDebugEventCallbacks::ChangeEngineState**	The engine state has changed.
DEBUG_EVENT_CHANGE_SYMBOL_STATE	**IDebugEventCallbacks::ChangeSymbolState**	The symbol state has changed.

Requirements

Header	DbgEng.h (include DbgEng.h)

DEBUG_FILTER_XXX

7/15/2019 • 2 minutes to read • Edit Online

The DEBUG_FILTER_*XXX* constants are used for three different purposes: to specify individual specific event filters, to specify the break status of an event filter, and to specify the handling status of an exception filter.

Specific Event Filter

The following constants are used to specify specific event filters.

VALUE	EVENT
DEBUG_FILTER_CREATE_THREAD	Create Thread
DEBUG_FILTER_EXIT_THREAD	Exit Thread
DEBUG_FILTER_CREATE_PROCESS	Create Process
DEBUG_FILTER_EXIT_PROCESS	Exit Process
DEBUG_FILTER_LOAD_MODULE	Load Module
DEBUG_FILTER_UNLOAD_MODULE	Unload Module
DEBUG_FILTER_SYSTEM_ERROR	System Error
DEBUG_FILTER_INITIAL_BREAKPOINT	Initial Break Point
DEBUG_FILTER_INITIAL_MODULE_LOAD	Initial Module Load
DEBUG_FILTER_DEBUGGEE_OUTPUT	Target Output

Break Status

The following constants are used to specify the break status of an event filter.

VALUE	DESCRIPTION
DEBUG_FILTER_BREAK	The event will break into the debugger.
DEBUG_FILTER_SECOND_CHANCE_BREAK	The event will break into the debugger if it is a second chance exception.

VALUE	DESCRIPTION
DEBUG_FILTER_OUTPUT	A notification of the event will be printed to the debugger console.
DEBUG_FILTER_IGNORE	The event is ignored.

Additionally, for an arbitrary exception filter, setting the break status to DEBUG_FILTER_REMOVE, removes the event filter.

Handling Status

The following constants are used to specify the handling status of an exception filter.

VALUE	DESCRIPTION
DEBUG_FILTER_GO_HANDLED	The exception has been handled.
DEBUG_FILTER_GO_NOT_HANDLED	The exception has not been handled.

Requirements

Header	DbgEng.h (include DbgEng.h)

DEBUG_FORMAT_XXX

10/23/2019 • 2 minutes to read • Edit Online

The DEBUG_FORMAT_XXX bit-flags are used by WriteDumpFile2 and WriteDumpFileWide to determine the format of a crash dump file and, for user-mode Minidumps, what information to include in the file.

The following bit-flags apply to all crash dump files.

VALUE	DESCRIPTION
DEBUG_FORMAT_WRITE_CAB	Package the crash dump file in a CAB file. The supplied file name or file handle is used for the CAB file; the crash dump is first created in a temporary file before being moved into the CAB file.
DEBUG_FORMAT_CAB_SECONDARY_FILES	Include the current symbols and mapped images in the CAB file. If DEBUG_FORMAT_WRITE_CAB is not set, this flag is ignored.
DEBUG_FORMAT_NO_OVERWRITE	Do not overwrite existing files.

The following bit-flags can also be included for user-mode Minidumps.

VALUE	DESCRIPTION
DEBUG_FORMAT_USER_SMALL_FULL_MEMORY	Add full memory data. All accessible committed pages owned by the target application will be included.
DEBUG_FORMAT_USER_SMALL_HANDLE_DATA	Add data about the handles that are associated with the target application.
DEBUG_FORMAT_USER_SMALL_UNLOADED_MODULES	Add unloaded module information. This information is available only in Windows Server 2003 and later versions of Windows.
DEBUG_FORMAT_USER_SMALL_INDIRECT_MEMORY	Add indirect memory. A small region of memory that surrounds any address that is referenced by a pointer on the stack or backing store is included.
DEBUG_FORMAT_USER_SMALL_DATA_SEGMENTS	Add all data segments within the executable images.
DEBUG_FORMAT_USER_SMALL_FILTER_MEMORY	Set to zero all of the memory on the stack and in the backing store that is not useful for recreating the stack trace. This can make compression of the Minidump more efficient and increase privacy by removing unnecessary information.
DEBUG_FORMAT_USER_SMALL_FILTER_PATHS	Remove the module paths, leaving only the module names. This is useful for protecting privacy by hiding the directory structure (which may contain the user's name).
DEBUG_FORMAT_USER_SMALL_FILTER_TRIAGE	This format is used to filter out any data that is not a pointer to other data captured in the dump. The flag can be used to reduce the amount of private data present in the dump while still allowing crashes to be diagnosed.
DEBUG_FORMAT_USER_SMALL_PROCESS_THREAD_DATA	Add the process environment block (PEB) and thread environment block (TEB). This flag can be used to provide Windows system information for threads and processes.
DEBUG_FORMAT_USER_SMALL_PRIVATE_READ_WRITE_MEMORY	Add all committed private read-write memory pages.
DEBUG_FORMAT_USER_SMALL_NO_OPTIONAL_DATA	Prevent privacy-sensitive data from being

included in the Minidump. Currently, this flag excludes from the Minidump data that would have been added due to the following flags being set: DEBUG_FORMAT_USER_SMALL_PROCESS_THREAD_DATA, DEBUG_FORMAT_USER_SMALL_FULL_ME

	MO RY, DEB UG_ FOR MAT _US ER_S MA LL_I NDI REC T_M EM ORY , DEB UG_ FOR MAT _US ER_S MA LL_P RIV ATE_ REA D_ WRI TE_ ME MO RY.
DEBUG_FORMAT_USER_SMALL_FULL_MEMORY_INFO	Add all basic memory information. This is the information returned by the IDebugDataSpaces2::QueryVirtual method. The information for all memory is included, not just valid memory, which allows the debugger to reconstruct the complete virtual memory layout from the Minidump.
DEBUG_FORMAT_USER_SMALL_THREAD_INFO	Add additional thread information, which includes execution time, start time, exit time, start address, and exit status.
DEBUG_FORMAT_USER_SMALL_CODE_SEGMENTS	Add all code segments with the executable images.

Requirements

Header	DbgEng.h (include DbgEng.h)

DEBUG_OUTCB_XXX

12/5/2018 • 2 minutes to read • Edit Online

The DEBUG_OUTCB_XXX constants are output flags. The output flags form a bit field that indicates the type of the output that accompanies them.

DEBUG_OUTCB_XXX defines different kinds of output callback notifications that can be sent to Output2.

The possible values include the following.

CONSTANT	DESCRIPTION
DEBUG_OUTCB_TEXT	Plain text content, flags are below, argument is mask.
DEBUG_OUTCB_DML	Debugger markup content, flags are below, argument is mask.
DEBUG_OUTCB_EXPLICIT_FLUSH	Notification of an explicit output flush, flags and argument are zero.

Requirements

Header	DbgEng.h (include DbgEng.h)

DEBUG_OUTCTL_XXX

6/25/2019 • 2 minutes to read • Edit Online

The DEBUG_OUTCTL_*XXX* constants are used for output control. The constants form a bit field that specifies the current policy of where to send output. The bit field is divided into two sections.

The lower bits must be exactly one of the following values.

VALUE	DESCRIPTION
DEBUG_OUTCTL_THIS_CLIENT	Output generated by methods called by this client will be sent only to this client's output callbacks.
DEBUG_OUTCTL_ALL_CLIENTS	Output will be sent to all clients.
DEBUG_OUTCTL_ALL_OTHER_CLIENTS	Output will be sent to all clients (except to the client that generated the output).
DEBUG_OUTCTL_IGNORE	Output will be discarded immediately and will not be logged or sent to callbacks.
DEBUG_OUTCTL_LOG_ONLY	Output will be logged but not sent to callbacks.

The higher bits of the bit field may contain the following values.

VALUE	DESCRIPTION
DEBUG_OUTCTL_NOT_LOGGED	Do not put output from this client in the global log file.
DEBUG_OUTCTL_OVERRIDE_MASK	Sends output to clients regardless of whether the client's output mask allows it.
DEBUG_OUTCTL_DML	For output that supports Debugger Markup Language (DML), sends the output in DML format.

To create a valid output control bit-field, take exactly one value from the first table, along with zero or more values from the second table, and combine them by using the bitwise-OR operator.

The default value of the output control bit-field is DEBUG_OUTCTL_ALL_CLIENTS.

As an alternative to creating your own output control bit-field, you can use one of the following values.

VALUE	DESCRIPTION

VALUE	DESCRIPTION
DEBUG_OUTCTL_AMBIENT_DML	Sets the new output control to the same value as the current output control and specifies that the output will be in DML format.
DEBUG_OUTCTL_AMBIENT_TEXT	Sets the new output control to the same value as the current output control and specifies that the output will be in text format.
DEBUG_OUTCTL_AMBIENT	Same as DEBUG_OUTCTL_AMBIENT_TEXT.

Requirements

Header	DbgEng.h (include DbgEng.h)

DEBUG_OUTPUT_XXX

12/5/2018 • 2 minutes to read • Edit Online

The DEBUG_OUTPUT_*XXX* constants are output flags. The output flags form a bit field that indicates the type of the output that accompanies them.

The possible values include the following.

CONSTANT	DESCRIPTION
DEBUG_OUTPUT_NORMAL	Normal output.
DEBUG_OUTPUT_ERROR	Error output.
DEBUG_OUTPUT_WARNING	Warnings.
DEBUG_OUTPUT_VERBOSE	Additional output.
DEBUG_OUTPUT_PROMPT	Prompt output.
DEBUG_OUTPUT_PROMPT_REGISTERS	Register dump before prompt.
DEBUG_OUTPUT_EXTENSION_WARNING	Warnings specific to extension operation.
DEBUG_OUTPUT_DEBUGGEE	Debug output from the target (for example, **OutputDebugString** or **DbgPrint**).
DEBUG_OUTPUT_DEBUGGEE_PROMPT	Debug input expected by the target (for example, **DbgPrompt**).
DEBUG_OUTPUT_SYMBOLS	Symbol messages (for example, **!sym noisy**).
DEBUG_OUTPUT_STATUS	Output which modifies the status bar.

Requirements

Header	DbgEng.h (include DbgEng.h)

DEBUG_PROCESS_XXX

6/25/2019 • 2 minutes to read • Edit Online

The process options are a bit set that control how the debugger engine treats user-modeprocesses. Some of these process options are global; others are specific to a process.

The process options only apply to live user-mode debugging.

BIT-FLAG	DESCRIPTION
DEBUG_PROCESS_DETACH_ON_EXIT	The debugger automatically detaches itself from the target process when the debugger exits. This is a global setting.
DEBUG_PROCESS_ONLY_THIS_PROCESS	The debugger will not debug child processes that are created by this process.

Requirements

Header	DbgEng.h (include DbgEng.h)

See also

.childdbg

DEBUG_STATUS_XXX

12/5/2018 • 2 minutes to read • Edit Online

The DEBUG_STATUS_*XXX* status codes have two purposes. They instruct the engine on how execution in the target should proceed, and they are used by the engine to report the execution status of the target.

After an event occurs, the engine can receive several instructions that tell it how execution in the target should proceed. In this case, it acts on the instruction with the highest precedence. Typically, the higher precedence status codes represent less execution for the target.

The values in the following table are reverse ordered by precedence; the values that appear earlier in the table have higher precedence.

STATUS CODE	WHEN REPORTING	WHEN INSTRUCTING	PRECEDENCE
DEBUG_STATUS_NO_DEBUGGEE	No debugging session is active.	N/A	
DEBUG_STATUS_OUT_OF_SYNC	The debugger communications channel is out of sync.	N/A	
DEBUG_STATUS_WAIT_INPUT	The target is awaiting input from the user.	N/A	
DEBUG_STATUS_TIMEOUT	The debugger communications channel has timed out.	N/A	
DEBUG_STATUS_BREAK	The target is suspended.	Suspend the target.	Highest precedence
DEBUG_STATUS_STEP_INTO	The target is executing a single instruction.	Continue execution of the target for a single instruction.	
DEBUG_STATUS_STEP_BRANCH	The target is executing until the next branch instruction.	Continue execution of the target until the next branch instruction.	
DEBUG_STATUS_STEP_OVER	The target is executing a single instruction or--if that instruction is a subroutine call--subroutine.	Continue execution of the target for a single instruction. If the instruction is a subroutine call, the call is entered and the target is allowed to run until the subroutine returns.	

STATUS CODE	WHEN REPORTING	WHEN INSTRUCTING	PRECEDENCE
DEBUG_STATUS_GO_NOT_HANDLED	N/A	Continue execution of the target, flagging the event as not handled.	
DEBUG_STATUS_GO_HANDLED	N/A	Continue execution of the target, flagging the event as handled.	
DEBUG_STATUS_GO	The target is executing normally.	Continue normal execution of the target.	
DEBUG_STATUS_IGNORE_EVENT	N/A	Continue previous execution of the target, ignoring the event.	
DEBUG_STATUS_RESTART_REQUESTED	The target is restarting.	Restart the target.	
DEBUG_STATUS_NO_CHANGE	N/A	No instruction. This value is returned by an event callback method when it does not wish to instruct the engine how to proceed with execution in the target.	Lowest precedence

> [!Note] > The precedence of the status codes does not follow the numeric values of the constants.

Requirements

Header	DbgEng.h (include DbgEng.h)

DEBUG_SYMBOL_XXX

10/23/2019 • 2 minutes to read • Edit Online

The DEBUG_SYMBOL_*XXX* constants are used for the symbol flags bit-set. The symbol flags describe (in part) a symbol in a symbol group.

The least significant bits of the symbol flags--the bits found in DEBUG_SYMBOL_EXPANSION_LEVEL_MASK--form a number that represents the expansion depth of the symbol within the symbol group. The depth of a child symbol is always one more than the depth of its parent symbol. For example, to find the depth of a symbol whose flags are contained in the variable *flags*, use the following statement:

```
depth = flags & DEBUG_SYMBOL_EXPANSION_LEVEL_MASK;
```

The rest of the symbol flags' bit-set can contain the following bit-flags.

CONSTANT	DESCRIPTION
DEBUG_SYMBOL_EXPANDED	The children of the symbol are part of the symbol group.
DEBUG_SYMBOL_READ_ONLY	The symbol represents a read-only variable.
DEBUG_SYMBOL_IS_ARRAY	The symbol represents an array variable.
DEBUG_SYMBOL_IS_FLOAT	The symbol represents a floating-point variable.
DEBUG_SYMBOL_IS_ARGUMENT	The symbol represents an argument passed to a function.
DEBUG_SYMBOL_IS_LOCAL	The symbol represents a local variable in a scope.

Requirements

Header	DbgEng.h (include DbgEng.h)

See also

DEBUG_SYMBOL_PARAMETERS

DEBUG_TYPEOPTS_XXX

6/25/2019 • 2 minutes to read • Edit Online

The type options affect how the engine formats numbers and strings for output.

The options are represented by a bit-set with the following bit flags.

CONSTANT	DESCRIPTION
DEBUG_TYPEOPTS_UNICODE_DISPLAY	When this bit is set, USHORT pointers and arrays are output as Unicode characters. This is equivalent to the debugger command **.enable_unicode 1**.
DEBUG_TYPEOPTS_LONGSTATUS_DISPLAY	When this bit is set, LONG integers are output in the default base instead of decimal. This is equivalent to the debugger command **.enable_long_status 1**.
DEBUG_TYPEOPTS_FORCERADIX_OUTPUT	When this bit is set, integers (except for LONG integers) are output in the default base instead of decimal. This is equivalent to the debugger command **.force_radix_output 1**.

Remarks

By default, all of the type formatting options are turned off.

For more information about types, see Types.

Requirements

Header	DbgEng.h (include DbgEng.h)

HRESULT Values

6/25/2019 • 2 minutes to read • Edit Online

The following is a list of common return values for functions and methods, and their usual meanings.

Successful results. These values are defined in WinError.h.

S_OK
Successful completion.

S_FALSE
Completed without error, but only partial results were obtained.

If a buffer is not large enough to hold the information that is returned to it, the returned information is often truncated to fit into the buffer and S_FALSE is returned from the method.

Error results. These values are defined in WinError.h.

E_FAIL
The operation could not be performed.

E_INVALIDARG
One of the arguments passed in was invalid.

E_NOINTERFACE
The object searched for was not found.

E_OUTOFMEMORY
A memory allocation attempt failed.

E_UNEXPECTED
The target was not accessible, or the engine was not in a state where the function or method could be processed.

E_NOTIMPL
Not implemented.

HRESULT_FROM_WIN32(ERROR_ACCESS_DENIED)
The operation was denied because the debugger is in Secure Mode.

NT error results. Other error codes, such as STATUS_CONTROL_C_EXIT and STATUS_NO_MORE_ENTRIES, can sometimes occur. These results are passed to the HRESULT_FROM_NT macro that is defined in WinError.h before being returned.

Win32 error results. Other error codes, such as ERROR_READ_FAULT and ERROR_WRITE_FAULT, can sometimes occur. These results are passed to the HRESULT_FROM_WIN32 macro that is defined in WinError.h before being returned.

Requirements

Header	DbgEng.h (include DbgEng.h)

IDebugAdvanced2::Request method

10/23/2019 • 2 minutes to read • Edit Online

The **Request** method performs a variety of different operations.

Syntax

```
HRESULT Request(
  [in]            ULONG  Request,
  [in, optional]  PVOID  InBuffer,
  [in]            ULONG  InBufferSize,
  [out, optional] PVOID  OutBuffer,
  [in]            ULONG  OutBufferSize,
  [out, optional] PULONG OutSize
);
```

Parameters

Request [in]
Specifies which operation to perform. **Request** can be one of the values in the following table. Details of each operation can be found by following the link in the "Request" column.

REQUEST	ACTION
DEBUG_REQUEST_SOURCE_PATH_HAS_SOURCE_SERVER	Check the source path for a source server.
DEBUG_REQUEST_TARGET_EXCEPTION_CONTEXT	Return the thread context for the stored event in a user-mode minidump file.
DEBUG_REQUEST_TARGET_EXCEPTION_THREAD	Return the operating system thread ID for the stored event in a user-mode minidump file.
DEBUG_REQUEST_TARGET_EXCEPTION_RECORD	Return the exception record for the stored event in a user-mode minidump file.
DEBUG_REQUEST_GET_ADDITIONAL_CREATE_OPTIONS	Return the default process creation options.
DEBUG_REQUEST_SET_ADDITIONAL_CREATE_OPTIONS	Set the default process creation options.
DEBUG_REQUEST_GET_WIN32_MAJOR_MINOR_VERSIONS	Return the version of Windows that is currently running on the target.

REQUEST	ACTION
DEBUG_REQUEST_READ_USER_MINIDUMP_STREAM	Read a stream from a user-mode minidump target.
DEBUG_REQUEST_TARGET_CAN_DETACH	Check to see if it is possible for the debugger engine to detach from the current process (leaving the process running but no longer being debugged).
DEBUG_REQUEST_SET_LOCAL_IMPLICIT_COMMAND_LINE	Set the debugger engine's implicit command line.
DEBUG_REQUEST_GET_CAPTURED_EVENT_CODE_OFFSET	Return the current event's instruction pointer.
DEBUG_REQUEST_READ_CAPTURED_EVENT_CODE_STREAM	Return up to 64 bytes of memory at the current event's instruction pointer.
DEBUG_REQUEST_EXT_TYPED_DATA_ANSI	Perform a variety of different operations that aid in the interpretation of typed data.

InBuffer [in, optional]
Specifies the input to this method. The type and interpretation of the input depends on the *Request* parameter.

InBufferSize [in]
Specifies the size of the input buffer *InBuffer*. If the request requires no input, *InBufferSize* should be set to zero.

OutBuffer [out, optional]
Receives the output from this method. The type and interpretation of the output depends on the *Request* parameter. If *OutBuffer* is **NULL**, the output is not returned.

OutBufferSize [in]
Specifies the size of the output buffer *OutBufferSize*. If the type of the output returned to *OutBuffer* has a known size, *OutBufferSize* is usually expected to be exactly that size, even if *OutBuffer* is set to **NULL**.

OutSize [out, optional]
Receives the size of the output returned in the output buffer *OutBuffer*. If *OutSize* is **NULL**, this information is not returned.

Return value

The interpretation of the return value depends on the value of the *Request* parameter. Unless otherwise stated, the following values may be returned.

RETURN CODE	DESCRIPTION
S_OK	The method was successful.

RETURN CODE	DESCRIPTION
S_FALSE	The method was successful. However, the output would not fit in the output buffer *OutBuffer*, so truncated output was returned.
E_INVALIDARG	The size of the input buffer *InBufferSize* or the size of the output buffer *OutBufferSize* was not the expected value.

This method may also return error values. See **Return Values** for more details.

Requirements

Target platform	Desktop
Header	Dbgeng.h (include Dbgeng.h)

See also

IDebugAdvanced2

IDebugAdvanced3

DEBUG_REQUEST_SOURCE_PATH_HAS_SOURCE_SERVER

DEBUG_REQUEST_TARGET_EXCEPTION_CONTEXT

DEBUG_REQUEST_TARGET_EXCEPTION_THREAD

DEBUG_REQUEST_TARGET_EXCEPTION_RECORD

DEBUG_REQUEST_GET_ADDITIONAL_CREATE_OPTIONS

DEBUG_REQUEST_SET_ADDITIONAL_CREATE_OPTIONS

DEBUG_REQUEST_GET_WIN32_MAJOR_MINOR_VERSIONS

DEBUG_REQUEST_READ_USER_MINIDUMP_STREAM

DEBUG_REQUEST_TARGET_CAN_DETACH

DEBUG_REQUEST_SET_LOCAL_IMPLICIT_COMMAND_LINE

DEBUG_REQUEST_GET_CAPTURED_EVENT_CODE_OFFSET

DEBUG_REQUEST_READ_CAPTURED_EVENT_CODE_STREAM

DEBUG_REQUEST_SOURCE_PATH_HAS_SOURCE_SERVER

12/5/2018 • 2 minutes to read • Edit Online

The DEBUG_REQUEST_SOURCE_PATH_HAS_SOURCE_SERVER **Request** operation checks the source path for a source server.

Parameters

InBuffer
Not used.

OutBuffer
Not used.

Return Value

S_OK
The source path includes a source server

S_FALSE
The source path does not include a source server.

See also

Request

DEBUG_REQUEST_TARGET_EXCEPTION_CONTEXT

10/23/2019 • 2 minutes to read • Edit Online

The DEBUG_REQUEST_TARGET_EXCEPTION_CONTEXT **Request** operation returns the thread context for the stored event in a user-mode minidump file.

Parameters

InBuffer
Not used.

OutBuffer
The thread context for the stored event. The type of the thread context is the CONTEXT structure for the target's effective processor at the time of the event. *OutBuffer* must be large enough to hold this structure.

Remarks

This information is also returned to the *Context* parameter by the **GetStoredEventInformation** method.

See also

Request

GetStoredEventInformation

DEBUG_REQUEST_TARGET_EXCEPTION_THREAD

12/5/2018 • 2 minutes to read • Edit Online

The DEBUG_REQUEST_TARGET_EXCEPTION_THREAD **Request** operation returns the operating system thread ID for the stored event in a user-mode minidump file.

Parameters

InBuffer
Not used.

OutBuffer
The operating system thread ID for the stored event. The type of the thread ID is ULONG.

See also

Request

DEBUG_REQUEST_TARGET_EXCEPTION_RECORD

12/5/2018 • 2 minutes to read • Edit Online

The DEBUG_REQUEST_TARGET_EXCEPTION_RECORD **Request** operation returns the exception record for the stored event in a user-mode minidump file.

InBuffer
Not used.

OutBuffer
The exception record for the stored event. The type of the exception record is EXCEPTION_RECORD64, which is defined in winnt.h.

See also

Request

DEBUG_REQUEST_GET_ADDITIONAL_CREATE_OPTIONS control code

10/23/2019 • 2 minutes to read • Edit Online

The DEBUG_REQUEST_GET_ADDITIONAL_CREATE_OPTIONS **Request** operation returns the default process creation options.

Parameters

InBuffer
Not used.

OutBuffer
The default process creation options. The type of the process creation options is **DEBUG_CREATE_PROCESS_OPTIONS**.

Remarks

The default process creation options are used by methods **CreateProcess** and **CreateProcessAndAttach** which, unlike **CreateProcess2** and **CreateProcessAndAttach2**, do not specify the full range of process creation options.

The **CreateFlags** field of the **DEBUG_CREATE_PROCESS_OPTIONS** structure is not used as a default because all process creation operations provide this information.

See also

Request

DEBUG_REQUEST_SET_ADDITIONAL_CREATE_OPTIONS

DEBUG_CREATE_PROCESS_OPTIONS

CreateProcess

CreateProcessAndAttach

DEBUG_REQUEST_SET_ADDITIONAL_CREATE_OPTIONS

10/23/2019 • 2 minutes to read • Edit Online

The DEBUG_REQUEST_SET_ADDITIONAL_CREATE_OPTIONS **Request** operation sets the default process creation options.

Parameters

InBuffer
The new default process creation options. The type of the process creation options is **DEBUG_CREATE_PROCESS_OPTIONS**.

OutBuffer
Not used.

Remarks

The default process creation options are used by methods **CreateProcess** and **CreateProcessAndAttach** which, unlike **CreateProcess2** and **CreateProcessAndAttach2**, do not specify the full range of process creation options.

The **CreateFlags** field of the **DEBUG_CREATE_PROCESS_OPTIONS** structure is not used as a default because all process creation operations provide this information.

See also

Request

DEBUG_REQUEST_GET_ADDITIONAL_CREATE_OPTIONS

DEBUG_CREATE_PROCESS_OPTIONS

CreateProcess

CreateProcessAndAttach

DEBUG_REQUEST_GET_WIN32_MAJOR_MINOR_VERSIONS

12/5/2018 • 2 minutes to read • Edit Online

The DEBUG_REQUEST_GET_WIN32_MAJOR_MINOR_VERSIONS **Request** operation returns the version of Windows that is currently running on the target.

Parameters

InBuffer
Not used.

OutBuffer
The major and minor version numbers for the version of Windows that is currently running on the target. The type of the version numbers is an array containing two ULONG values; the first ULONG in the array is the major version number and the second is the minor version number

The following table lists the major and minor version numbers by operating system.

VERSION OF WINDOWS	MAJOR VERSION NUMBER	MINOR VERSION NUMBER
Windows 2000	5	0
Windows XP	5	1
Windows Server 2003	5	2

See also

Request

DEBUG_REQUEST_TARGET_CAN_DETACH

12/5/2018 • 2 minutes to read • Edit Online

The DEBUG_REQUEST_TARGET_CAN_DETACH **Request** operation checks to see if it is possible for the debugger engine to detach from the current process (leaving the process running but no longer being debugged).

Parameters

InBuffer
Not used.

OutBuffer
Not used.

Return Value

S_OK
It is possible to detach the debugger from the current process.

S_FALSE
It is not possible to detach the debugger from the current process.

Remarks

Only targets running on Microsoft Windows XP or later versions of Windows support detaching the debugger from the process.

See also

Request

DEBUG_REQUEST_SET_LOCAL_IMPLICIT_COMMAND_LINE

10/23/2019 • 2 minutes to read • Edit Online

The DEBUG_REQUEST_SET_LOCAL_IMPLICIT_COMMAND_LINE **Request** operation sets the debugger engine's implicit command line.

Parameters

InBuffer
The new implicit command line. The type of *InBuffer* is a pointer to a Unicode string (PWSTR). The pointer is copied but the string it points to is not copied.

OutBuffer
Not used.

Remarks

The implicit command line can be used as the command line when creating a process. The process creation options (**DEBUG_CREATE_PROCESS_OPTIONS**) contain an option for using the implicit command line instead of a supplied command line when creating a process.

See also

Request

DEBUG_CREATE_PROCESS_OPTIONS

DEBUG_REQUEST_GET_ADDITIONAL_CREATE_OPTIONS

DEBUG_REQUEST_SET_ADDITIONAL_CREATE_OPTIONS

CreateProcess2

CreateProcessAndAttach2

DEBUG_REQUEST_GET_CAPTURED_EVENT_CODE_OFFSET

6/25/2019 • 2 minutes to read • Edit Online

The DEBUG_REQUEST_GET_CAPTURED_EVENT_CODE_OFFSET **Request** operation returns the current event's instruction pointer.

Parameters

InBuffer
Not used.

OutBuffer
The instruction pointer of the current event. The type of the instruction pointer is ULONG64.

Return Value

S_OK
The method was successful.

E_NOINTERFACE
The memory at the instruction pointer for the current event is not valid.

This method may also return error values. See **Return Values** for more details.

Remarks

The memory at the instruction pointer for the current event can be read using the **Request** operation's **DEBUG_REQUEST_READ_CAPTURED_EVENT_CODE_STREAM**.

See also

Request

DEBUG_REQUEST_READ_CAPTURED_EVENT_CODE_STREAM

DEBUG_REQUEST_READ_CAPTURED_EVENT_CODE_STREAM

12/5/2018 • 2 minutes to read • Edit Online

The DEBUG_REQUEST_READ_CAPTURED_EVENT_CODE_STREAM **Request** operation returns up to 64 bytes of memory at the current event's instruction pointer.

Parameters

InBuffer
Not used.

OutBuffer
The memory at the current event's instruction pointer. Up to 64 bytes of memory may be returned.

Remarks

The memory returned is a snapshot of the memory taken when the event occurred. It does not reflect any changes that may have been made to the target's memory since the event.

The current event's instruction pointer is returned by the **Request** operation **DEBUG_REQUEST_GET_CAPTURED_EVENT_CODE_OFFSET**.

See also

Request

DEBUG_REQUEST_GET_CAPTURED_EVENT_CODE_OFFSET

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI

10/23/2019 • 2 minutes to read • Edit Online

The DEBUG_REQUEST_EXT_TYPED_DATA_ANSI **Request** operation performs a variety of different sub-operations that aid in the interpretation of typed data.

Parameters

InBuffer
Specifies the **EXT_TYPED_DATA** structure that determines the sub-operation to perform. This EXT_TYPED_DATA structure contains the input parameters for that sub-operation along with any (optional) additional data. The additional data is included in *InBuffer* after the EXT_TYPED_DATA structure. The size of *InBuffer* is the total size of the buffer that contains the EXT_TYPED_DATA structure and the additional data. See **EXT_TYPED_DATA** for details on this structure and how to include the additional data.

The following sub-operations are supported.

SUB-OPERATION	DESCRIPTION
EXT_TDOP_COPY	Makes a copy of a typed data description.
EXT_TDOP_RELEASE	Releases a typed data description.
EXT_TDOP_SET_FROM_EXPR	Returns the value of an expression.
EXT_TDOP_SET_FROM_U64_EXPR	Returns the value of an expression. An optional address can be provided as a parameter to the expression.
EXT_TDOP_GET_FIELD	Returns a member of a structure.
EXT_TDOP_EVALUATE	Returns the value of an expression. An optional value can be provided as a parameter to the expression.
EXT_TDOP_GET_TYPE_NAME	Returns the type name for typed data.
EXT_TDOP_OUTPUT_TYPE_NAME	Prints the type name for typed data.
EXT_TDOP_OUTPUT_SIMPLE_VALUE	Prints the value of typed data.
EXT_TDOP_OUTPUT_FULL_VALUE	Prints the type and value for typed data.
EXT_TDOP_HAS_FIELD	Determines if a structure contains a specified member.

SUB-OPERATION	DESCRIPTION
EXT_TDOP_GET_FIELD_OFFSET	Returns the offset of a member within a structure.
EXT_TDOP_GET_ARRAY_ELEMENT	Returns an element from an array.
EXT_TDOP_GET_DEREFERENCE	Dereferences a pointer, returning the value it points to.
EXT_TDOP_GET_TYPE_SIZE	Returns the size of the specified typed data.
EXT_TDOP_OUTPUT_TYPE_DEFINITION	Prints the definition of the type for the specified typed data.
EXT_TDOP_GET_POINTER_TO	Returns a new typed data description that represents a pointer to specified typed data.
EXT_TDOP_SET_FROM_TYPE_ID_AND_U64	Creates a typed data description from a type and memory location.
EXT_TDOP_SET_PTR_FROM_TYPE_ID_AND_U64	Creates a typed data description that represents a pointer to a specified memory location with specified type.

OutBuffer
Receives the **EXT_TYPED_DATA** structure that contains the output parameters and any additional data for the sub-operation. As with *InBuffer*, the size of *OutBuffer* is the total size of the buffer that contains the EXT_TYPED_DATA structure and any additional data.

The DEBUG_REQUEST_EXT_TYPED_DATA_ANSI operation will initially copy *InBuffer* into *OutBuffer* and then modify the contents of *OutBuffer* in place. This means that *OutBuffer* will be populated with the input parameters of the EXT_TYPED_DATA and any additional data that was provided in *InBuffer*. It also means that the size of *OutBuffer* must be at least as big as the size of *InBuffer*.

Return Value

S_OK
The operation was successful.

This method can also return error values. See **Return Values** for more details.

The value returned by this operation is also stored in the **Status** member of *OutBuffer*.

Remarks

The sub-operation performed by the DEBUG_REQUEST_EXT_TYPED_DATA_ANSI **Request** operation is determined by the **Operation** member of the **EXT_TYPED_DATA** structure, which takes a value in the **EXT_TDOP** enumeration.

See also

EXT_TYPED_DATA

EXT_TDOP

Request

EXT_TDOP_COPY

10/23/2019 • 2 minutes to read • Edit Online

The EXT_TDOP_COPY sub-operation of the **DEBUG_REQUEST_EXT_TYPED_DATA_ANSIRequest** operation makes a copy of a typed data description.

Parameters

Operation
Set to EXT_TDOP_COPY for this sub-operation.

InData
Specifies the original instance of the **DEBUG_TYPED_DATA** structure.

OutData
Receives the copy of the instance of the **DEBUG_TYPED_DATA** structure specified by **InData**.

Status
Receives the status code returned by this sub-operation. This is the same as the value returned by **Request**.

Remarks

EXT_TDOP_COPY is a value in the **EXT_TDOP** enumeration.

The parameters for this sub-operation are members of the **EXT_TYPED_DATA** structure. The members of EXT_TYPED_DATA that are not listed in the preceding Parameters section are not used by this sub-operation and should be set to zero. The descriptions of the members in the preceding Parameters section specify what the members are used for. For more details on the members, see **EXT_TYPED_DATA**.

See also

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI

EXT_TDOP

EXT_TYPED_DATA

Request

EXT_TDOP_RELEASE

10/23/2019 • 2 minutes to read • Edit Online

The EXT_TDOP_RELEASE sub-operation of the **DEBUG_REQUEST_EXT_TYPED_DATA_ANSIRequest** operation releases a typed data description.

Parameters

Operation
Set to EXT_TDOP_RELEASE for this sub-operation.

InData
Specifies the instance of the **DEBUG_TYPED_DATA** structure to release.

Status
Receives the status code returned by this sub-operation. This is the same as the value returned by **Request**.

Remarks

EXT_TDOP_RELEASE is a value in the **EXT_TDOP** enumeration.

The parameters for this sub-operation are members of the **EXT_TYPED_DATA** structure. The members of EXT_TYPED_DATA that are not listed in the preceding Parameters section are not used by this sub-operation and should be set to zero. The descriptions of the members in the preceding Parameters section specify only the purpose of the members in this particular setting. See **EXT_TYPED_DATA** for more details.

See also

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI

EXT_TDOP

EXT_TYPED_DATA

Request

EXT_TDOP_SET_FROM_EXPR

10/23/2019 • 2 minutes to read • Edit Online

The EXT_TDOP_SET_FROM_EXPR sub-operation of the **DEBUG_REQUEST_EXT_TYPED_DATA_ANSIRequest** operation returns a typed data description that represents the value of an expression.

Parameters

Operation
Set to EXT_TDOP_SET_FROM_EXPR for this sub-operation.

Flags
Specifies the bit flags that describe the target's memory in which the value of the expression resides. See **EXT_TYPED_DATA** for details of these flags.

OutData
Receives the **DEBUG_TYPED_DATA** structure that represents the value of the expression.

InStrIndex
Specifies the expression to evaluate. This expression is evaluated by the default expression evaluator.

Status
Receives the status code returned by this sub-operation. This is the same as the value returned by **Request**.

Remarks

EXT_TDOP_SET_FROM_EXPR is a value in the **EXT_TDOP** enumeration.

The parameters for this sub-operation are members of the **EXT_TYPED_DATA** structure. The members of EXT_TYPED_DATA that are not listed in the preceding Parameters section are not used by this sub-operation and should be set to zero. The descriptions of the members in the preceding Parameters section specify what the members are used for. See **EXT_TYPED_DATA** for more details.

See also

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI

EXT_TDOP

EXT_TYPED_DATA

Request

EXT_TDOP_SET_FROM_U64_EXPR

10/23/2019 • 2 minutes to read • Edit Online

The EXT_TDOP_SET_FROM_U64_EXPR sub-operation of the **DEBUG_REQUEST_EXT_TYPED_DATA_ANSIRequest** operation returns a typed data description that represents the value of an expression.

Operation
Set to EXT_TDOP_SET_FROM_U64_EXPR for this sub-operation.

Flags
Specifies the bit flags that describe the target's memory in which the value of the expression resides. See **EXT_TYPED_DATA** for details of these flags.

InData
Specifies optional typed data whose address in the target's memory can be used in the expression specified by **InStrIndex**. This address is used by the expression as the pseudo-register **$extin**.

OutData
Receives the **DEBUG_TYPED_DATA** structure that represents the value of the expression.

InStrIndex
Specifies the expression to evaluate. This expression is evaluated by the default expression evaluator.

Status
Receives the status code returned by this sub-operation. This is the same as the value returned by **Request**.

Remarks

EXT_TDOP_SET_FROM_U64_EXPR is a value in the **EXT_TDOP** enumeration.

The parameters for this sub-operation are members of the **EXT_TYPED_DATA** structure. The members of EXT_TYPED_DATA that are not listed in the preceding Parameters section are not used by this sub-operation and should be set to zero. The descriptions of the members in the preceding Parameters section specify what the members are used for. See **EXT_TYPED_DATA** for more details.

See also

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI

EXT_TDOP

EXT_TYPED_DATA

Request

EXT_TDOP_GET_FIELD

10/23/2019 • 2 minutes to read • Edit Online

The EXT_TDOP_GET_FIELD sub-operation of the **DEBUG_REQUEST_EXT_TYPED_DATA_ANSIRequest** operation returns typed data description that represents a member of a given structure.

Parameters

Operation
Set to EXT_TDOP_GET_FIELD for this sub-operation.

InData
Specifies an instance of **DEBUG_TYPED_DATA** that describes the structure whose member is desired.

OutData
Receives an instance of **DEBUG_TYPED_DATA** for the requested member.

InStrIndex
Specifies the name of the requested member. The name of the member is a dot-separated path and can contain sub-members - for example, **mymember.mysubmember**. Pointers on this dot-separated path will automatically be dereferenced. However, a dot operator (**.**) should still be used here instead of the usual C pointer dereference operator (**->**).

Status
Receives the status code returned by this sub-operation. This is the same as the value returned by **Request**.

Remarks

EXT_TDOP_GET_FIELD is a value in the **EXT_TDOP** enumeration.

The parameters for this sub-operation are members of the **EXT_TYPED_DATA** structure. The members of EXT_TYPED_DATA that are not listed in the preceding Parameters section are not used by this sub-operation and should be set to zero. The descriptions of the members in the preceding Parameters section specify what the members are used for. See **EXT_TYPED_DATA** for more details.

See also

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI

EXT_TDOP

EXT_TYPED_DATA

Request

EXT_TDOP_EVALUATE

10/23/2019 • 2 minutes to read • Edit Online

The EXT_TDOP_EVALUATE sub-operation of the **DEBUG_REQUEST_EXT_TYPED_DATA_ANSIRequest** operation returns the typed data description that represents the value of an expression.

Parameters

Operation
Set to EXT_TDOP_EVALUATE for this sub-operation.

Flags
Specifies the bit flags that describe the target's memory in which the value of the expression resides. See **EXT_TYPED_DATA** for details of these flags.

InData
Specifies optional typed data whose value can be used in the expression specified by **InStrIndex**. This value is used by the expression as the pseudo-register **$extin**.

OutData
Receives the **DEBUG_TYPED_DATA** structure that represents the value of the expression.

InStrIndex
Specifies the expression to evaluate. This expression is evaluated by the default expression evaluator.

Status
Receives the status code returned by this sub-operation. This is the same as the value returned by **Request**.

Remarks

EXT_TDOP_EVALUATE is a value in the **EXT_TDOP** enumeration.

The parameters for this sub-operation are members of the **EXT_TYPED_DATA** structure. The members of EXT_TYPED_DATA that are not listed in the preceding Parameters section are not used by this sub-operation and should be set to zero. The descriptions of the members in the preceding Parameters section specify what the members are used for. See **EXT_TYPED_DATA** for more details.

See also

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI

EXT_TDOP

EXT_TYPED_DATA

Request

EXT_TDOP_GET_TYPE_NAME

10/23/2019 • 2 minutes to read • Edit Online

The EXT_TDOP_GET_TYPE_NAME sub-operation of the **DEBUG_REQUEST_EXT_TYPED_DATA_ANSIRequest** operation returns the type name of the specified typed data.

Parameters

Operation
Set to EXT_TDOP_GET_TYPE_NAME for this sub-operation.

InData
Specifies the typed data whose type name is being requested.

StrBufferIndex
Receives the type name.

StrBufferChars
Specifies the size, in characters, of the ANSI string buffer **StrBufferIndex**.

StrCharsNeeded
Receives the size, in characters, of the type name.

Status
Receives the status code returned by this sub-operation. This is the same as the value returned by **Request**.

Remarks

EXT_TDOP_GET_TYPE_NAME is a value in the **EXT_TDOP** enumeration.

The parameters for this sub-operation are members of the **EXT_TYPED_DATA** structure. The members of EXT_TYPED_DATA that are not listed in the preceding Parameters section are not used by this sub-operation and should be set to zero. The descriptions of the members in the preceding Parameters section specify what the members are used for. See **EXT_TYPED_DATA** for more details.

See also

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI

EXT_TDOP

EXT_TYPED_DATA

Request

EXT_TDOP_OUTPUT_TYPE_NAME

10/23/2019 • 2 minutes to read • Edit Online

The EXT_TDOP_OUTPUT_TYPE_NAME sub-operation of the **DEBUG_REQUEST_EXT_TYPED_DATA_ANSIRequest** operation prints the type name of the specified typed data.

Parameters

Operation
Set to EXT_TDOP_OUTPUT_TYPE_NAME for this sub-operation.

InData
Specifies the typed data whose type name will be printed.

Status
Receives the status code returned by this sub-operation. This is the same as the value returned by **Request**.

Remarks

The type name is sent to the debugger engine's output callbacks.

EXT_TDOP_OUTPUT_TYPE_NAME is a value in the **EXT_TDOP** enumeration.

The parameters for this sub-operation are members of the **EXT_TYPED_DATA** structure. The members of EXT_TYPED_DATA that are not listed in the preceding Parameters section are not used by this sub-operation and should be set to zero. The descriptions of the members in the preceding Parameters section specify what the members are used for. See **EXT_TYPED_DATA** for more details.

See also

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI

EXT_TDOP

EXT_TYPED_DATA

Request

EXT_TDOP_OUTPUT_SIMPLE_VALUE

10/23/2019 • 2 minutes to read • Edit Online

The EXT_TDOP_OUTPUT_SIMPLE_VALUE sub-operation of the **DEBUG_REQUEST_EXT_TYPED_DATA_ANSIRequest** operation prints the value of the specified typed data.

Parameters

Operation
Set to EXT_TDOP_OUTPUT_SIMPLE_VALUE for this sub-operation.

InData
Specifies the typed data whose value will be printed.

Status
Receives the status code returned by this sub-operation. This is the same as the value returned by **Request**.

Remarks

The value is formatted and is sent to the debugger engine's output callbacks.

EXT_TDOP_OUTPUT_SIMPLE_VALUE is a value in the **EXT_TDOP** enumeration.

The parameters for this sub-operation are members of the **EXT_TYPED_DATA** structure. The members of EXT_TYPED_DATA that are not listed in the preceding Parameters section are not used by this sub-operation and should be set to zero. The descriptions of the members in the preceding Parameters section specify what the members are used for. See **EXT_TYPED_DATA** for more details.

See also

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI

EXT_TDOP

EXT_TYPED_DATA

Request

EXT_TDOP_OUTPUT_FULL_VALUE

10/23/2019 • 2 minutes to read • Edit Online

The EXT_TDOP_OUTPUT_FULL_VALUE sub-operation of the **DEBUG_REQUEST_EXT_TYPED_DATA_ANSIRequest** operation prints the type and value of the specified typed data.

Parameters

Operation
Set to EXT_TDOP_OUTPUT_FULL_VALUE for this sub-operation.

InData
Specifies the typed data whose type name and value will be printed.

Status
Receives the status code returned by this sub-operation. This is the same as the value returned by **Request**.

Remarks

The type name and formatted value are sent to the debugger engine's output callbacks. EXT_TDOP_OUTPUT_FULL_VALUE prints more detailed information about the value than **EXT_TDOP_OUTPUT_SIMPLE_VALUE**. For example, pointers are dereferenced and the values they point to are also printed.

EXT_TDOP_OUTPUT_FULL_VALUE is a value in the **EXT_TDOP** enumeration.

The parameters for this sub-operation are members of the **EXT_TYPED_DATA** structure. The members of EXT_TYPED_DATA that are not listed in the preceding Parameters section are not used by this sub-operation and should be set to zero. The descriptions of the members in the preceding Parameters section specify what the members are used for. See **EXT_TYPED_DATA** for more details.

See also

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI

EXT_TDOP

EXT_TYPED_DATA

Request

EXT_TDOP_HAS_FIELD

10/23/2019 • 2 minutes to read • Edit Online

The EXT_TDOP_HAS_FIELD sub-operation of the **DEBUG_REQUEST_EXT_TYPED_DATA_ANSIRequest** operation determines if a structure contains a specified member.

Parameters

Operation
Set to EXT_TDOP_HAS_FIELD for this sub-operation.

InData
Specifies the typed data that is checked for the existence of the member. The typed data is first checked to see if it represents an instance of a structure, then the structure is checked to see if it contains the specified member.

InStrIndex
Specifies the name of the member. The name is a dot-separated path and can contain sub-members - for example, **mymember.mysubmember**. Pointers on this dot-separated path will automatically be dereferenced. However, a dot operator (**.**) should still be used here instead of the usual C pointer dereference operator (**->**).

Status
Receives the status code returned by this sub-operation. This is the same as the value returned by **Request**.

If the typed data contains the member, **Status** receives S_OK. If the typed data does not contain the member, **Status** receives E_NOINTERFACE. Other error values might also be returned.

Remarks

EXT_TDOP_HAS_FIELD is a value in the **EXT_TDOP** enumeration.

The parameters for this sub-operation are members of the **EXT_TYPED_DATA** structure. The members of EXT_TYPED_DATA that are not listed in the preceding Parameters section are not used by this sub-operation and should be set to zero. The descriptions of the members in the preceding Parameters section specify what the members are used for. See **EXT_TYPED_DATA** for more details.

See also

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI

EXT_TDOP

EXT_TYPED_DATA

Request

EXT_TDOP_GET_FIELD_OFFSET

10/23/2019 • 2 minutes to read • Edit Online

The EXT_TDOP_GET_FIELD_OFFSET sub-operation of the **DEBUG_REQUEST_EXT_TYPED_DATA_ANSIRequest** operation returns the offset of a member within a structure.

Parameters

Operation
Set to EXT_TDOP_GET_FIELD_OFFSET for this sub-operation.

InData
Specifies the typed data that represents an instance of a structure that contains the member whose offset is being requested.

InStrIndex
Specifies the name of the member whose offset is being requested. The name is a dot-separated path and can contain sub-members. For example, **mymember.mysubmember**. Pointers on this dot-separated path will automatically be dereferenced. However, a dot operator (**.**) should still be used here instead of the usual C pointer dereference operator (**->**).

Out32
Receives the offset of the member within an instance of the structure. This is the number of bytes between the beginning of an instance of the structure and the member.

Status
Receives the status code returned by this sub-operation. This is the same as the value returned by **Request**.

Remarks

EXT_TDOP_GET_FIELD_OFFSET is a value in the **EXT_TDOP** enumeration.

The parameters for this sub-operation are members of the **EXT_TYPED_DATA** structure. The members of EXT_TYPED_DATA that are not listed in the preceding Parameters section are not used by this sub-operation and should be set to zero. The descriptions of the members in the preceding Parameters section specify what the members are used for. See **EXT_TYPED_DATA** for more details.

See also

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI

EXT_TDOP

EXT_TYPED_DATA

Request

EXT_TDOP_GET_ARRAY_ELEMENT

10/23/2019 • 2 minutes to read • Edit Online

The EXT_TDOP_GET_ARRAY_ELEMENT sub-operation of the **DEBUG_REQUEST_EXT_TYPED_DATA_ANSIRequest** operation returns an element from an array.

Parameters

Operation
Set to EXT_TDOP_GET_ARRAY_ELEMENT for this sub-operation.

InData
Specifies the array whose element is being requested. **InData** can also specify a pointer, in which case it is treated as an array.

OutData
Receives the requested element from the array.

In64
Specifies the index of the element in the array.

Status
Receives the status code returned by this sub-operation. This is the same as the value returned by **Request**.

Remarks

EXT_TDOP_GET_ARRAY_ELEMENT is a value in the **EXT_TDOP** enumeration.

The parameters for this sub-operation are members of the **EXT_TYPED_DATA** structure. The members of EXT_TYPED_DATA that are not listed in the preceding Parameters section are not used by this sub-operation and should be set to zero. The descriptions of the members in the preceding Parameters section specify what the members are used for. See **EXT_TYPED_DATA** for more details.

See also

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI

EXT_TDOP

EXT_TYPED_DATA

Request

EXT_TDOP_GET_DEREFERENCE

10/23/2019 • 2 minutes to read • Edit Online

The EXT_TDOP_GET_DEREFERENCE sub-operation of the **DEBUG_REQUEST_EXT_TYPED_DATA_ANSIRequest** operation dereferences a pointer and returns the value it points to.

Parameters

Operation
Set to EXT_TDOP_GET_DEREFERENCE for this sub-operation.

InData
Specifies the pointer to dereference. **InData** can also specify an array, in which case the first element in the array is returned.

OutData
Receives the value pointed to.

Status
Receives the status code returned by this sub-operation. This is the same as the value returned by **Request**.

Remarks

EXT_TDOP_GET_DEREFERENCE is a value in the **EXT_TDOP** enumeration.

The parameters for this sub-operation are members of the **EXT_TYPED_DATA** structure. The members of EXT_TYPED_DATA that are not listed in the preceding Parameters section are not used by this sub-operation and should be set to zero. The descriptions of the members in the preceding Parameters section specify what the members are used for. See **EXT_TYPED_DATA** for more details.

See also

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI

EXT_TDOP

EXT_TYPED_DATA

Request

EXT_TDOP_GET_TYPE_SIZE

10/23/2019 • 2 minutes to read • Edit Online

The EXT_TDOP_GET_TYPE_SIZE sub-operation of the **DEBUG_REQUEST_EXT_TYPED_DATA_ANSIRequest** operation returns the size of the specified typed data.

Parameters

Operation
Set to EXT_TDOP_GET_TYPE_SIZE for this sub-operation.

InData
Specifies the typed data whose size is being requested.

Out32
Receives the size, in bytes, of the typed data.

Status
Receives the status code returned by this sub-operation. This is the same as the value returned by **Request**.

Remarks

EXT_TDOP_GET_TYPE_SIZE is a value in the **EXT_TDOP** enumeration.

The parameters for this sub-operation are members of the **EXT_TYPED_DATA** structure. The members of EXT_TYPED_DATA that are not listed in the preceding Parameters section are not used by this sub-operation and should be set to zero. The descriptions of the members in the preceding Parameters section specify what the members are used for. See **EXT_TYPED_DATA** for more details.

See also

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI

EXT_TDOP

EXT_TYPED_DATA

Request

EXT_TDOP_OUTPUT_TYPE_DEFINITION

10/23/2019 • 2 minutes to read • Edit Online

The EXT_TDOP_OUTPUT_TYPE_DEFINITION sub-operation of the **DEBUG_REQUEST_EXT_TYPED_DATA_ANSIRequest** operation prints the definition of the type for the specified typed data.

Parameters

Operation
Set to EXT_TDOP_OUTPUT_TYPE_DEFINITION for this sub-operation.

InData
Specifies the typed data whose type's definition will be printed.

Status
Receives the status code returned by this sub-operation. This is the same as the value returned by **Request**.

Remarks

The definition of the type is formatted and sent to the debugger engine's output callbacks.

EXT_TDOP_OUTPUT_TYPE_DEFINITION is a value in the **EXT_TDOP** enumeration.

The parameters for this sub-operation are members of the **EXT_TYPED_DATA** structure. The members of EXT_TYPED_DATA that are not listed in the preceding Parameters section are not used by this sub-operation and should be set to zero. The descriptions of the members in the preceding Parameters section specify what the members are used for. See **EXT_TYPED_DATA** for more details.

See also

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI

EXT_TDOP

EXT_TYPED_DATA

Request

EXT_TDOP_GET_POINTER_TO

10/23/2019 • 2 minutes to read • Edit Online

The EXT_TDOP_GET_POINTER_TO sub-operation of the **DEBUG_REQUEST_EXT_TYPED_DATA_ANSIRequest** operation returns a new typed data description that represents a pointer to specified typed data.

Parameters

Operation
Set to EXT_TDOP_GET_POINTER_TO for this sub-operation.

InData
Specifies the original typed data description to which a pointer is returned.

OutData
Receives a typed data description that represents a pointer to the typed data specified by **InData**.

Status
Receives the status code returned by this sub-operation. This is the same as the value returned by **Request**.

Remarks

EXT_TDOP_GET_POINTER_TO is a value in the **EXT_TDOP** enumeration.

The parameters for this sub-operation are members of the **EXT_TYPED_DATA** structure. The members of EXT_TYPED_DATA that are not listed in the preceding Parameters section are not used by this sub-operation and should be set to zero. The descriptions of the members in the preceding Parameters section specify what the members are used for. See **EXT_TYPED_DATA** for more details.

See also

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI

EXT_TDOP

EXT_TYPED_DATA

Request

EXT_TDOP_SET_FROM_TYPE_ID_AND_U64

10/23/2019 • 2 minutes to read • Edit Online

The EXT_TDOP_SET_FROM_TYPE_ID_AND_U64 sub-operation of the **DEBUG_REQUEST_EXT_TYPED_DATA_ANSIRequest** operation creates a typed data description from a data type and a memory location.

Parameters

Operation
Set to EXT_TDOP_SET_FROM_TYPE_ID_AND_U64 for this sub-operation.

Flags
Specifies the bit flags that describe the target's memory in which the value of the typed data resides. See **EXT_TYPED_DATA** for details of these flags.

InData
Specifies the type and the memory location. This instance of the **DEBUG_TYPED_DATA** structure can be manually created and populated with the required members. The following members are used:

ModBase
Specifies the location in the target's virtual memory of the base address of the module that contains the type.

Offset
Specifies the location in the target's memory of the data. **Offset** is a virtual memory address unless there are flags present in **Flags** that specify that **Offset** is a physical memory address.

TypeId
Specifies the type ID of the type.

OutData
Receives the typed data description.

Status
Receives the status code returned by this sub-operation. This is the same as the value returned by **Request**.

Remarks

EXT_TDOP_SET_FROM_TYPE_ID_AND_U64 is a value in the **EXT_TDOP** enumeration.

The parameters for this sub-operation are members of the **EXT_TYPED_DATA** structure. The members of EXT_TYPED_DATA that are not listed in the preceding Parameters section are not used by this sub-operation and should be set to zero. The descriptions of the members in the preceding Parameters section specify what the members are used for. See **EXT_TYPED_DATA** for more details.

See also

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI

EXT_TDOP

EXT_TYPED_DATA

Request

EXT_TDOP_SET_PTR_FROM_TYPE_ID_AND_U64

10/23/2019 • 2 minutes to read • Edit Online

The EXT_TDOP_SET_PTR_FROM_TYPE_ID_AND_U64 sub-operation of the **DEBUG_REQUEST_EXT_TYPED_DATA_ANSIRequest** operation creates a typed data description that represents a pointer to a specified memory location with a specified type.

Parameters

Operation
Set to EXT_TDOP_SET_PTR_FROM_TYPE_ID_AND_U64 for this sub-operation.

Flags
Specifies the bit flags that describe the target's memory in which the value of the typed data resides. See **EXT_TYPED_DATA** for details of these flags.

InData
Specifies the type and memory location. This instance of the **DEBUG_TYPED_DATA** structure can be manually created and populated with the required members. The following members are used:

ModBase
Specifies the location in the target's virtual memory of the base address of the module that contains the type.

Offset
Specifies the location in the target's memory of the data. **Offset** is a virtual memory address unless there are flags present in **Flags** that specify that **Offset** is a physical memory address.

TypeId
Specifies the type ID of the type.

OutData
Receives the typed data description that represents a pointer to the memory location and type.

Status
Receives the status code returned by this sub-operation. This is the same as the value returned by **Request**.

Remarks

EXT_TDOP_SET_PTR_FROM_TYPE_ID_AND_U64 is a value in the **EXT_TDOP** enumeration.

The parameters for this sub-operation are members of the **EXT_TYPED_DATA** structure. The members of EXT_TYPED_DATA that are not listed in the preceding Parameters section are not used by this sub-operation and should be set to zero. The descriptions of the members in the preceding Parameters section specify what the members are used for. See **EXT_TYPED_DATA** for more details.

See also

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI

EXT_TDOP

EXT_TYPED_DATA

Request

ExtExtension

10/23/2019 • 8 minutes to read • Edit Online

The **ExtExtension** class is the base class for the C++ class that represents the EngExtCpp extension library.

The **ExtExtension** class includes the following methods, which can be used by the subclass:

Initialize

Uninitialize

OnSessionActive

OnSessionInactive

OnSessionAccessible

OnSessionInaccessible

IsUserMode

IsKernelMode

IsLiveLocalUser

IsMachine32

IsCurMachine32

IsMachine64

IsCurMachine64

Is32On64

CanQueryVirtual

HasFullMemBasic

IsExtensionRemote

AreOutputCallbacksDmlAware

RequireUserMode

RequireKernelMode

GetNumUnnamedArgs

GetUnnamedArgStr

GetUnnamedArgU64

HasUnnamedArg

GetArgStr

GetArgU64

HasArg

HasCharArg

SetUnnamedArg

SetUnnamedArgStr

SetUnnamedArgU64

SetArg

SetArgStr

SetArgU64

GetRawArgStr

GetRawArgCopy

Out

Warn

Err

Verb

Dml

DmlWarn

DmlErr

DmlVerb

DmlCmdLink

DmlCmdExec

RefreshOutputCallbackFlags

WrapLine

OutWrapStr

OutWrapVa

OutWrap

DemandWrap

AllowWrap

TestWrap

RequestCircleString

CopyCircleString

PrintCircleStringVa

PrintCircleString

SetAppendBuffer

AppendBufferString

AppendStringVa

AppendString

IsAppendStart

SetCallStatus

GetCachedSymbolTypeId

GetCachedFieldOffset

GetCachedFieldOffset

AddCachedSymbolInfo

GetExpr64

GetExprU64

GetExprS64

ThrowCommandHelp

ThrowInterrupt

ThrowOutOfMemory

ThrowContinueSearch

ThrowReloadExtension

ThrowInvalidArg

ThrowRemote

ThrowStatus

ThrowLastError

The **ExtExtension** class also contains the following fields that can be used by the subclass:

```
class ExtExtension
{
public:
    USHORT  m_ExtMajorVersion;
    USHORT  m_ExtMinorVersion;
    ULONG  m_ExtInitFlags;
    ExtKnownStruct *  m_KnownStructs;
    ExtProvidedValue *  m_ProvidedValues;
    ExtCheckedPointer<IDebugAdvanced>  m_Advanced;
    ExtCheckedPointer<IDebugClient>  m_Client;
    ExtCheckedPointer<IDebugControl>  m_Control;
    ExtCheckedPointer<IDebugDataSpaces>  m_Data;
    ExtCheckedPointer<IDebugRegisters>  m_Registers;
    ExtCheckedPointer<IDebugSymbols>  m_Symbols;
    ExtCheckedPointer<IDebugSystemObjects>  m_System;
    ExtCheckedPointer<IDebugAdvanced2>  m_Advanced2;
    ExtCheckedPointer<IDebugAdvanced3>  m_Advanced3;
    ExtCheckedPointer<IDebugClient2>  m_Client2;
    ExtCheckedPointer<IDebugClient3>  m_Client3;
    ExtCheckedPointer<IDebugClient4>  m_Client4;
    ExtCheckedPointer<IDebugClient5>  m_Client5;
    ExtCheckedPointer<IDebugControl2>  m_Control2;
    ExtCheckedPointer<IDebugControl3>  m_Control3;
    ExtCheckedPointer<IDebugControl4>  m_Control4;
    ExtCheckedPointer<IDebugDataSpaces2>  m_Data2;
    ExtCheckedPointer<IDebugDataSpaces3>  m_Data3;
    ExtCheckedPointer<IDebugDataSpaces4>  m_Data4;
    ExtCheckedPointer<IDebugRegisters2>  m_Registers2;
    ExtCheckedPointer<IDebugSymbols2>  m_Symbols2;
    ExtCheckedPointer<IDebugSymbols3>  m_Symbols3;
    ExtCheckedPointer<IDebugSystemObjects2>  m_System2;
    ExtCheckedPointer<IDebugSystemObjects3>  m_System3;
    ExtCheckedPointer<IDebugSystemObjects4>  m_System4;
    ULONG  m_OutputWidth;
    ULONG  m_ActualMachine;
    ULONG  m_Machine;
    ULONG  m_PageSize;
    ULONG  m_PtrSize;
    ULONG  m_NumProcessors;
    ULONG64  m_OffsetMask;
    ULONG  m_DebuggeeClass;
    ULONG  m_DebuggeeQual;
    ULONG  m_DumpFormatFlags;
    bool  m_IsRemote;
    bool  m_OutCallbacksDmlAware;
    ULONG  m_OutMask;
    ULONG  m_CurChar;
    ULONG  m_LeftIndent;
    bool  m_AllowWrap;
    bool  m_TestWrap;
    ULONG  m_TestWrapChars;
    PSTR  m_AppendBuffer;
    ULONG  m_AppendBufferChars;
    PSTR  m_AppendAt;
};
```

Members

m_ExtMajorVersion

The major version number of the extension library. This should be set by the **Initialize** method. If it is not set, it defaults to **1**.

m_ExtMinorVersion

The minor version number of the extension library. This should be set by the **Initialize** method. If it is not set, it

defaults to **0** (zero).

m_ExtInitFlags
The DbgEng extension initialization flags for *DebugExtensionInitialize*.

m_KnownStructs
An array of **ExtKnownStruct** structures that the extension library is capable of formatting for output. This member should be set by the **Initialize** method and should not be changed once this method returns.

If **m_KnownStructs** is not **NULL**, the **TypeName** member of the last **ExtKnownStruct** structure in the array must be **NULL**.

When formatting a target's structure for output, if the name of the structure's type matches the **TypeName** member of one of the **ExtKnownStruct** structures in this array, the callback function specified in the **Method** member is called to perform the formatting.

m_ProvidedValues
An array of **ExtProvidedValue** structures listing the pseudo registers that the extension library can provide values for. This member should be set by the **Initialize** method and should not be changed once this method returns.

If **m_ProvidedValues** is not **NULL**, the **ValueName** member of the last **ExtProvidedValue** structure in the array must be **NULL**.

When evaluating a pseudo register, if the name of the pseudo register matches the **ValueName** member of one of the **ExtProvidedValue** structures in this array, the callback function specified in the **Method** member is called to evaluate the pseudo register.

m_Advanced
The **IDebugAdvanced** interface pointer for the client object that can be used by the extension library. It is valid during the invocation of externally-called extension methods-for example, the execution of an extension command, a call to *ExtKnownStructMethod* and **ExtProvideValueMethod**.

m_Client
The **IDebugClient** interface pointer for the client object that can be used by the extension library. It is valid during the invocation of externally-called extension methods-for example, the execution of an extension command, a call to *ExtKnownStructMethod* and **ExtProvideValueMethod**.

m_Control
The **IDebugControl** interface pointer for the client object that can be used by the extension library. It is valid during the invocation of externally-called extension methods-for example, the execution of an extension command, a call to *ExtKnownStructMethod* and **ExtProvideValueMethod**.

m_Data
The **IDebugDataSpaces** interface pointer for the client object that can be used by the extension library. It is valid during the invocation of externally-called extension methods-for example, the execution of an extension command, a call to *ExtKnownStructMethod* and **ExtProvideValueMethod**.

m_Registers
The **IDebugRegisters** interface pointer for the client object that can be used by the extension library. It is valid during the invocation of externally-called extension methods-for example, the execution of an extension command, a call to *ExtKnownStructMethod* and **ExtProvideValueMethod**.

m_Symbols
The **IDebugSymbols** interface pointer for the client object that can be used by the extension library. It is valid during the invocation of externally-called extension methods-for example, the execution of an extension command, a call to *ExtKnownStructMethod* and **ExtProvideValueMethod**.

m_System

The **IDebugSystemObjects** interface pointer for the client object that can be used by the extension library. It is valid during the invocation of externally-called extension methods-for example, the execution of an extension command, a call to *ExtKnownStructMethod* and **ExtProvideValueMethod**.

m_Advanced2
The **IDebugAdvanced2** interface pointer for the client object that can be used by the extension library. It is valid during the invocation of externally-called extension methods-for example, the execution of an extension command, a call to *ExtKnownStructMethod* and **ExtProvideValueMethod**. This interface might not be available in all versions of the debugger engine.

m_Advanced3
The **IDebugAdvanced3** interface pointer for the client object that can be used by the extension library. It is valid during the invocation of externally-called extension methods-for example, the execution of an extension command, a call to *ExtKnownStructMethod* and **ExtProvideValueMethod**. This interface might not be available in all versions of the debugger engine.

m_Client2
The **IDebugClient2** interface pointer for the client object that can be used by the extension library. It is valid during the invocation of externally-called extension methods-for example, the execution of an extension command, a call to *ExtKnownStructMethod* and **ExtProvideValueMethod**. This interface might not be available in all versions of the debugger engine.

m_Client3
The **IDebugClient3** interface pointer for the client object that can be used by the extension library. It is valid during the invocation of externally-called extension methods-for example, the execution of an extension command, a call to *ExtKnownStructMethod* and **ExtProvideValueMethod**. This interface might not be available in all versions of the debugger engine.

m_Client4
The **IDebugClient4** interface pointer for the client object that can be used by the extension library. It is valid during the invocation of externally-called extension methods-for example, the execution of an extension command, a call to *ExtKnownStructMethod* and **ExtProvideValueMethod**. This interface might not be available in all versions of the debugger engine.

m_Client5
The **IDebugClient5** interface pointer for the client object that can be used by the extension library. It is valid during the invocation of externally-called extension methods-for example, the execution of an extension command, a call to *ExtKnownStructMethod* and **ExtProvideValueMethod**. This interface might not be available in all versions of the debugger engine.

m_Control2
The **IDebugControl2** interface pointer for the client object that can be used by the extension library. It is valid during the invocation of externally-called extension methods-for example, the execution of an extension command, a call to *ExtKnownStructMethod* and **ExtProvideValueMethod**. This interface might not be available in all versions of the debugger engine.

m_Control3
The **IDebugControl3** interface pointer for the client object that can be used by the extension library. It is valid during the invocation of externally-called extension methods-for example, the execution of an extension command, a call to *ExtKnownStructMethod* and **ExtProvideValueMethod**. This interface might not be available in all versions of the debugger engine.

m_Control4
The **IDebugControl4** interface pointer for the client object that can be used by the extension library. It is valid during the invocation of externally-called extension methods-for example, the execution of an extension command, a call to *ExtKnownStructMethod* and **ExtProvideValueMethod**. This interface might not be available in all

versions of the debugger engine.

m_Data2
The **IDebugDataSpaces2** interface pointer for the client object that can be used by the extension library. It is valid during the invocation of externally-called extension methods-for example, the execution of an extension command, a call to *ExtKnownStructMethod* and **ExtProvideValueMethod**. This interface might not be available in all versions of the debugger engine.

m_Data3
The **IDebugDataSpaces3** interface pointer for the client object that can be used by the extension library. It is valid during the invocation of externally-called extension methods-for example, the execution of an extension command, a call to *ExtKnownStructMethod* and **ExtProvideValueMethod**. This interface might not be available in all versions of the debugger engine.

m_Data4
The IDebugDataSpaces4 interface pointer for the client object that can be used by the extension library. It is valid during the invocation of externally-called extension methods-for example, the execution of an extension command, a call to *ExtKnownStructMethod* and **ExtProvideValueMethod**. This interface might not be available in all versions of the debugger engine.

m_Registers2
The **IDebugRegisters2** interface pointer for the client object that can be used by the extension library. It is valid during the invocation of externally-called extension methods-for example, the execution of an extension command, a call to *ExtKnownStructMethod* and **ExtProvideValueMethod**. This interface might not be available in all versions of the debugger engine.

m_Symbols2
The **IDebugSymbols2** interface pointer for the client object that can be used by the extension library. It is valid during the invocation of externally-called extension methods-for example, the execution of an extension command, a call to *ExtKnownStructMethod* and **ExtProvideValueMethod**. This interface might not be available in all versions of the debugger engine.

m_Symbols3
The **IDebugSymbols3** interface pointer for the client object that can be used by the extension library. It is valid during the invocation of externally-called extension methods-for example, the execution of an extension command, a call to *ExtKnownStructMethod* and **ExtProvideValueMethod**. This interface might not be available in all versions of the debugger engine.

m_System2
The **IDebugSystemObjects2** interface pointer for the client object that can be used by the extension library. It is valid during the invocation of externally-called extension methods-for example, the execution of an extension command, a call to *ExtKnownStructMethod* and **ExtProvideValueMethod**. This interface might not be available in all versions of the debugger engine.

m_System3
The **IDebugSystemObjects3** interface pointer for the client object that can be used by the extension library. It is valid during the invocation of externally-called extension methods-for example, the execution of an extension command, a call to *ExtKnownStructMethod* and **ExtProvideValueMethod**. This interface might not be available in all versions of the debugger engine.

m_System4
The **IDebugSystemObjects4** interface pointer for the client object that can be used by the extension library. It is valid during the invocation of externally-called extension methods-for example, the execution of an extension command, a call to *ExtKnownStructMethod* and **ExtProvideValueMethod**. This interface might not be available in all versions of the debugger engine.

m_PtrSize
The size of a pointer on the current target. If the target uses 32-bit pointers, **m_PtrSize** is 4. If the target uses 64-bit pointers, **m_PtrSize** is 8.

m_AppendBuffer
A character buffer used to return strings from the extension library to the engine. The size of this buffer is **m_AppendBufferChars**. The methods **AppendBufferString**, **AppendStringVa**, and **AppendString** can be used to write strings to this buffer.

m_AppendBufferChars
The size, in characters, of the buffer **m_AppendBuffer**.

Specific Exceptions

12/5/2018 • 2 minutes to read • Edit Online

The following table lists the exception codes for the specific exception filters.

EXCEPTION CODE	EXCEPTION	HEADER FILE OR VALUE
STATUS_ACCESS_VIOLATION	Access violation	NtStatus.h
STATUS_ASSERTION_FAILURE	Assertion failure	NtStatus.h
STATUS_APPLICATION_HANG	Application hang	0xCFFFFFFF
STATUS_BREAKPOINT	Break instruction exception	NtStatus.h
STATUS_CPP_EH_EXCEPTION	C++ exception handling exception	0xE06D7363
STATUS_CLR_EXCEPTION	Common language runtime (CLR) exception	0xE0434f4D
DBG_CONTROL_BREAK	CTRL+Break exception	NtStatus.h
DBG_CONTROL_C	CTRL+C exception	NtStatus.h
STATUS_DATATYPE_MISALIGNMENT	Data misaligned	NtStatus.h
DBG_COMMAND_EXCEPTION	Debugger command exception	NtStatus.h
STATUS_GUARD_PAGE_VIOLATION	Guard page violation	NtStatus.h
STATUS_ILLEGAL_INSTRUCTION	Illegal instruction	NtStatus.h
STATUS_IN_PAGE_ERROR	In-page I/O error	NtStatus.h
STATUS_INTEGER_DIVIDE_BY_ZERO	Integer divide-by-zero	NtStatus.h
STATUS_INTEGER_OVERFLOW	Integer overflow	NtStatus.h

EXCEPTION CODE	EXCEPTION	HEADER FILE OR VALUE
STATUS_INVALID_HANDLE	Invalid handle	NtStatus.h
STATUS_INVALID_LOCK_SEQUENCE	Invalid lock sequence	NtStatus.h
STATUS_INVALID_SYSTEM_SERVICE	Invalid system call	NtStatus.h
STATUS_PORT_DISCONNECTED	Port disconnected	NtStatus.h
STATUS_SINGLE_STEP	Single-step exception	NtStatus.h
STATUS_STACK_BUFFER_OVERRUN	Stack buffer overflow	NtStatus.h
STATUS_STACK_OVERFLOW	Stack overflow	NtStatus.h
STATUS_VERIFIER_STOP	Application Verifier stop	NtStatus.h
STATUS_WAKE_SYSTEM_DEBUGGER	Wake debugger	NtStatus.h

WdbgExts Functions

10/23/2019 • 2 minutes to read • Edit Online

The wdbgexts.h header file contains prototypes for the following functions. These functions use the same prototype for both 32-bit and 64-bit extensions:

GetContext

SetContext

CheckControlC

GetCurrentProcessAddr

GetCurrentProcessHandle

GetCurrentThreadAddr

GetDebuggerCacheSize

GetDebuggerData

Disasm

dprintf

GetExpression

GetExpressionEx

GetInputLine

Ioctl

GetKdContext

ReadMemory

SearchMemory

WriteMemory

ReadMsr

WriteMsr

GetPebAddress

ReadPhysical

ReadPhysicalWithFlags

WritePhysical

WritePhysicalWithFlags

GetTebAddress

StackTrace

GetSymbol

ReloadSymbols

GetSetSympath

TranslateVirtualToPhysical

The wdbgexts.h header file contains prototypes for the following functions. These functions have different prototypes for 32-bit and 64-bit extensions:

ReadControlSpace

ReadControlSpace64

ReadTypedControlSpace32

ReadTypedControlSpace64

WriteControlSpace

ReadIoSpace

ReadIoSpace64

ReadIoSpaceEx

ReadIoSpaceEx64

WriteIoSpace

WriteIoSpace64

WriteIoSpaceEx

WriteIoSpaceEx64

SetThreadForOperation

SetThreadForOperation64

The wdbgexts.h header file contains prototypes for the following functions. These functions can be used only in 64-bit extensions:

GetFieldData

GetFieldOffset

GetFieldValue

GetShortField

ReadField

ReadListEntry

ReadPointer

WritePointer

IsPtr64

ReadPtr

GetTypeSize

InitTypeRead

InitTypeReadPhysical

ListType

Customizing Debugger Output Using DML

10/23/2019 • 20 minutes to read • Edit Online

The debugger markup language (DML) provides a mechanism for enhancing output from the debugger and extensions. Similar to HTML, the debugger's markup support allows output to include display directives and extra non-display information in the form of tags. The debugger user interfaces, such as WinDbg parse out the extra information provided in DML to enhance the display of information and provide new behaviors, such as grid displays and sorting. This topic describes how you can customize your debug output using DML. For general information on enabling and using DML in the debuggers, see Using Debugger Markup Language.

DML is available in Windows 10 and later.

DML Overview

On of DML's primary benefits it to provide the ability to link to related information in debugger output. One of the primary DML tags is the <link> tag which lets an output producer indicate that information related to a piece of output can be accessed via the link's stated action. As with HTML links in a web browser, this allows the user to navigate hyperlinked information.

A benefit of providing hyperlinked content is that it can be used to enhance the discoverability of debugger and debugger extension functionality. The debugger and its extensions contain a large amount of functionality but it can be difficult to determine the appropriate command to use in different scenarios. Users must simply know what commands are available to use in specific scenarios. Differences between user and kernel debugging add further complexity. This often means that many users are unaware of debug commands which could help them. DML links provides the ability for arbitrary debug commands to be wrapped in alternate presentations, such as descriptive text, clickable menu systems or linked help. Using DML, the command output can be enhanced to guide the user to additional related commands relevant for the task at hand.

Debugger DML Support

- The command window in WinDbg supports all DML behavior and will show colors, font styles and links.
- The console debuggers – ntsd, cdb and kd – only support the color attributes of DML, and the only when running in a true console with color mode enabled.
- Debuggers with redirected I/O, ntsd –d or remote.exe sessions will not display any colors.

DML Content Specification

DML is not intended to be a full presentation language such as HTML. DML is deliberately very simple and has only a handful of tags.

Because not all debugger tools support rich text, DML is designed to allow simple translation between DML and plain text. This allows DML to function in all existing debugger tools. Effects such as colors can easily be supported since removing them does not remove the text carrying the actual information.

DML is not XML. DML does not attempt to carry semantic nor structured information. As mentioned above, there must be a simple mapping between DML and plain text, for this reason, DML tags are all discardable.

DML is not extensible; all tags are pre-defined and validated to work across all of the existing debugger tools.

Tag Structure

Similar to XML, DML tags are given as a starting <tagname [args]> and a following </tagname>.

Special Characters

DML content roughly follows the XML/HTML rules for special characters. The characters &, <, > and " are special and cannot be used in plain text. The equivalent escaped versions are &, <, > and ". For example this text:

"Alice & Bob think 3 < 4"

would be converted to the following DML.

```
"Alice & Bob think 3 &lt 4"
```

C programming language formatting characters

A significant departure from XML/HTML rules is that DML text can include C programming language stream-style formatting characters such as \b, \t, \r and \n. This is to support compatibility with existing debugger text production and consumption.

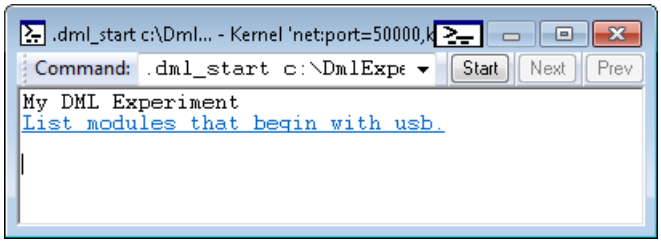
Example DML

Suppose the file C:\Dml_Experiment.txt contains the following lines.

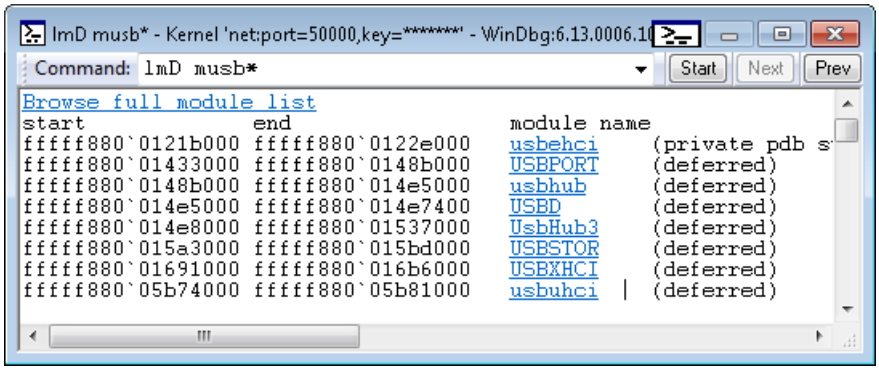
```
My DML Experiment
<link cmd="lmD musb*">List modules that begin with usb.</link>
```

The following command displays the text and link in the Command Browser window.

```
.browse .dml_start c:\Dml_Experiment.txt
```



If you click the **List modules that begin with usb** link, you see output similar to the following image.



Right-Click Behavior in DML

Right-click behavior is available in DML. This sample shows how to define right click behavior using <altlink> to send a breakpoint **bp (Set Breakpoint)** command and send the **u (Unassemble)** with a regular click.

```
<link cmd="u MyProgram!memcpy">
<altlink name="Set Breakpoint (bp)" cmd="bp MyProgram!memcpy" />
u MyProgram!memcpy
</link>
```

DML Tag Reference

<link>

<link [name="text"] [cmd="debugger_command"][alt="Hover text to display"] [section="name"]>link text</link>

The link tag is the basic hyper linking mechanism in DML. It directs user interfaces which support DML presentation to display the link text as a clickable link. When a link with a cmd specification is clicked the debugger command is executed and its output should replace the current output.

The name and section arguments allow for navigation between named links, similar to HTML's <a name> and #name support. When a link that has a section argument is clicked on the UI will scan for a link named with a matching name and will scroll that into view. This allows links to point to different sections of the same page (or a particular section of a new page). DML's section name is separate to avoid having to define a new syntax which would allow a section name at the end of the command string.

Conversion to plain text drops the tags.

Example

```
<b> Handy Links </b>
<link cmd="!dml_proc">Display process information with DML rendering.</link>
<link cmd="kM">Display stack information with DML rendering.</link>
```

Example

This example shows the use of the alt attribute to create text that will appear when you hover over the DML link.

```
<b>Hover Example</b>
<link cmd="lmD" alt="This link will run the list modules command and display the output in DML
format">LmD</link>
```

<altlink>

<altlink [name="text"] [cmd="debugger_command"] [section="name"]>alt link text</altlink>

The <altlink> tag provides right-click behavior is available in DML. When a link with a cmd specification is clicked the debugger command is executed and its output should replace the current output. The <altlink> tab is normally paired with the <link> tag to support regular and right click behavior.

Conversion to plain text drops the tags.

Example

This example shows how to define right click behavior using <altlink> to send a breakpoint **bp (Set Breakpoint)** command and send the **u (Unassemble)** with a regular click.

```
<link cmd="u MyProgram!memcpy">
<altlink name="Set Breakpoint (bp)" cmd="bp MyProgram!memcpy" />
u MyProgram!memcpy
</link>
```

<exec>

<exec cmd="debugger_command">descriptive text</exec>

An exec tag is similar to a link tag in that the descriptive text should be presented as a clickable item. However, when the exec tag is used in a command browser window, the given command is executed without replacing the current output, this tag provides a way to have commands executed with a click, from a menu.

Conversion to plain text drops the tags.

Example

This example shows how to define two commands with a regular click.

```
<b>Exec Sample</b>
<exec cmd="!dml_proc">Display process information with DML rendering.</exec>
<exec cmd="kM">Display stack information with DML rendering.</exec>
```

****

bold text

This tag requests bold. The , and <u> can be nested to have a mix of the properties.

Conversion to plain text drops the tags.

Example

This example shows how to bold text.

```
<b>This is bold Text</b>
```

****

italicized text

This tag requests italics. The , and <u> can be nested to have a mix of the properties.

Conversion to plain text drops the tags.

Example

This example shows how to italicize text.

```
<i>This is italicized Text</i>
```

<u>

<u>underlined text</u>

This tag requests underlined text. The , and <u> can be nested to have a mix of the properties.

Conversion to plain text drops the tags.

Example

This example shows how to underlined text.

```
<u>This is underlined Text</u>
```

Example

This example shows how to combine tags to bold, underline and italicize text.

```
<b><u><i>This is bold, underlined and italized text. </i></u></b>
```

<col>

<col fg="name" bg="name">text</col>

Request foreground and background colors for the text. The colors are given as names of known colors instead of absolute values as that allows customers to control what kind of color they see. Current color names (defaults only apply to WinDbg).

Foreground and Background Element Tags

Setting	Description / Example
wbg - Windows background wfg - Windows foreground	Default window background and foreground colors. Default to system colors for window and window text. <col fg="wfg" bg="wbg"> This is standard foreground / background text </col>
clbg - Current line foreground clfg - Current line background	Current line background and foreground colors. Default to system colors for highlight and highlight text. <col fg="clfg" bg="clbg"> Test Text - Current Line</col>
empbg - Emphasized background emphfg - Emphasized foreground	Emphasized text. Defaults to light blue. <col fg="empfg" bg="empbg"> This is emphasis foreground / background text </col>
subbg - Subdued background subfg- Subdued foreground	Subdued text. Default to system color for inactive caption text and inactive captions. <col fg="subfg" bg="subbg"> This is subdued foreground / background text </col>
normbg - Normal background normfg - Normal foreground	Normal <col fg="normfg" bg="normbg"> Test Text - Normal (normfg / normbg) </col>
warnbg - Warning background warnfg - Warning foreground	Warning <col fg="warnfg" bg="warnbg"> Test Text - Warning (warnfg / warnbg) </col>
errbg - Error background errfg - Error foreground	Error <col fg="errfg" bg="errbg"> Test Text - Error (errfg / errbg) </col>
verbbg - Verbose background verbfg - Verbose foreground	Verbose <col fg="verbfg" bg="verbbg"> Test Text - Verbose (verbfg / verbbg) </col>

Source Code Single Element Tags

srcnum - Source numeric constant	Source element colors. <col fg="srcnum" bg="wbg"> Test Text - srcnum </col>
srcchar - Source character constant	<col fg="srcchar" bg="wbg"> Test Text - srcchar </col>
srcstr - Source string constant	<col fg="srcstr" bg="wbg"> Test Text - srcstr </col>
srcid -Source identifier	<col fg="srcid " bg="wbg"> Test Text - srcid </col>
srckw- Keyword	<col fg="srckw" bg="wbg"> Test Text - srckw </col>
srcpair - Source brace or matching symbol pair	<col fg="srcpair" bg="empbbg"> Test Text - srcpair </col>
srccmnt - Source comment	<col fg="srccmnt" bg="wbg"> Test Text - srccmnt </col>
srcdrct - Source directive	<col fg="srcdrct" bg="wbg"> Test Text - srcdrct </col>
srcspid - Source special identifier	<col fg="srcspid" bg="wbg"> Test Text - srcspid </col>
srcannot - Source annotation	<col fg="srcannot" bg="wbg"> Test Text - srcannot </col>
changed - Changed data	Used for data that has changed since a previous stop point, such as changed registers in WinDbg. Defaults to red. <col fg="changed" bg="wbg"> Test Text - Changed</col>

DML Example Code

This example code illustrates the following.

- Calling debug commands
- Implementing right click commands
- Implementing hover over text
- Using color and rich text

```
<col fg="srckw" bg="wbg"> <b>
*****************************************************
*** Example debug commands for crash dump analysis ****
*****************************************************
</b></col>
<col fg="srcchar" bg="wbg"><i>
**** Hover over commands for additional information ****
        **** Right-click for command help ****
</i></col>

<col fg="srccmnt" bg="wbg"><b>*** Common First Steps for Crash Dump Analysis ***</b> </col>
```

<link cmd=".symfix" alt="Set standard symbol path using .symfix">.symfix<altlink name="Help about .symfix" cmd=".hh .symfix" /> </link> - Set standard symbol path
<link cmd=".sympath+ C:\Symbols" alt="This link adds addtional symbol directories">.sympath+ C:\Symbols<altlink name="Help for .sympath" cmd=".hh .sympath" /> </link> - Add any additional symbol directories, for example C:\Symbols
<link cmd=".reload /f" alt="This link reloads symbols">.reload /f<altlink name="Help for .reload" cmd=".hh .reload" /> </link> - Reloads symbols to make sure they are in good shape
<link cmd="!analyze -v" alt="This link runs !analyze with the verbose option">!analyze -v<altlink name="Help for !analyze" cmd=".hh !analyze" /> </link> - Run !analyze with the verbose option
<link cmd="vertarget" alt="This link runs checks the target version">vertarget<altlink name="Help for vertarget" cmd=".hh vertarget" /></link> - Check the target version
<link cmd="version" alt="This link displays the versions in use">version<altlink name="Help for version" cmd=".hh version" /></link> - Display the versions in use
<link cmd=".chain /D" alt="This link runs .chain">.chain /D<altlink name="Help for .chain" cmd=".hh .chain" /> </link> - Use the .chain /D command to list debugger extensions
<link cmd="kM" alt="This link displays the stack backtrace using DML">kD<altlink name="Help for k" cmd=".hh k, kb, kc, kd, kp, kP, kv (Display Stack Backtrace)" /> </link> - Display the stack backtrace using DML rendering
<link cmd="lmD" alt="This link will run the list modules command and display the output in DML format">LmD<altlink name="Help for lmD" cmd=".hh lm" /> </link> - List modules command and display the output in DML format
<link cmd=".help /D" alt="Display help for commands">.help /D <altlink name="Help for .dot commands" cmd=".hh commands" /></link> - Display help for commands in WinDbg
<link cmd=".hh" alt="Start help">.hh<altlink name="Debugger Reference Help".hh Contents" cmd=".hh Debugger Reference" /></link> - Start help

<col fg="srccmnt" bg="wbg">*** Registers and Context ***</col>
<link cmd="r" alt="This link displays registers">r<altlink name="Help about r command" cmd=".hh r" /></link> - Display registers
<link cmd="dt nt!_CONTEXT" alt="This link displays information about nt_CONTEXT">dt nt!_CONTEXT<altlink name="Help about the dt command" cmd=".hh dt" /></link> - Display information about nt_CONTEXT
<link cmd="dt nt!_PEB" alt="This link calls the dt command to display nt!_PEB">dt nt!_PEB<altlink name="Help about dt command" cmd=".hh dt" /></link> - Display information about the nt!_PEB
<link cmd="ub" alt="This link unassembles backwards">ub<altlink name="Help about ub command" cmd=".hh u, ub, uu (Unassemble)" /></link> - Unassemble Backwards

<col fg="srcchar" bg="wbg">
**** Note: Not all of the following commands will work with all crash dump data ****
</col>
<col fg="srccmnt" bg="wbg">*** Device Drivers ***</col>
<link cmd="!devnode 0 1" alt="This link displays the devnodes">!devnode 0 1<altlink name="Help about !devnode command" cmd=".hh !devnode" /></link> - Display devnodes
<link cmd=".load wdfkd.dll;!wdfkd.help" alt="Load wdfkd extensions and display help">.load wdfkd.dll;!wdfkd.help<altlink name="Help about the wdfkd extensions" cmd=".hh !wdfkd" /></link> - Load wdfkd extensions and display help
<link cmd="!wdfkd.wdfldr" alt="This link displays !wdfkd.wdfldr">!wdfkd.wdfldr<altlink name="Help about !wdfkd.wdfldr" cmd=".hh !wdfkd.wdfldr" /></link> - Display WDF framework driver loader information
<link cmd="!wdfkd.wdfumtriage" alt="This link displays !wdfkd.umtriage">!wdfkd.umtriage<altlink name="Help about !wdfkd.umtriage" cmd=".hh !wdfkd_wdfumtriage" /></link> - Display WDF umtriage driver information

<col fg="srccmnt" bg="wbg">*** IRPs and IRQL ***</col>
<link cmd="!processirps" alt="This link displays process IRPs">!processirps<altlink name="Help about !processirps command" cmd=".hh !processirps" /></link> - Display process IRPs
<link cmd="!irql" alt="This link displays !irql">!irql<altlink name="Help about !irql command" cmd=".hh !irql" /></link> - Run !irql

<col fg="srccmnt" bg="wbg">*** Variables and Symbols ***</col>
<link cmd="dv" alt="This link calls the dv command">dv<altlink name="Help about dv command" cmd=".hh dv" /> </link> - Display the names and values of all local variables in the current scope

<col fg="srccmnt" bg="wbg">*** Threads, Processes, and Stacks ***</col>
<link cmd="!threads" alt="This link displays threads">!threads<altlink name="Help about the !threads command" cmd=".hh !threads" /></link> - Display threads
<link cmd="!ready 0xF" alt="This link runs !ready 0xF">!ready 0xF<altlink name="Help about the !ready command" cmd=".hh !ready" /></link> - Display threads in the ready state
<link cmd="!process 0 F" alt="This link runs !process 0 F ">!process 0 F<altlink name="Help about the !process command" cmd=".hh !process" /></link> - Run !process 0 F
<link cmd="!stacks 2" alt="This link displays stack information using !stacks 2 ">!stacks 2<altlink name="Help about the !stacks command" cmd=".hh !stacks" /></link> - Display stack information using !stacks 2
<link cmd=".tlist" alt="This link displays a process list using TList ">tlist<altlink name="Help about the

```
TList command" cmd=".hh .tlist" /></link> - Display a process list using tlist
<link cmd="!process" alt="This link displays process ">!process<altlink name="Help about the !process command"
cmd=".hh !process" /></link> - Display process information
<link cmd="!dml_proc" alt="This link displays process information with DML rendering.">!dml_proc<altlink
name="Help about the !dml_proc command" cmd=".hh !dml_proc" /></link> - Display process information with DML
rendering
```

This example code illustrates the use of color and formatting tags.

```
*** Text Tag Examples ****

<b>This is bold text</b>
<u>This is underlined text</u>
<i>This is italizized text</i>
<b><u><i>This is bold, underlined and italizized text</i></u></b>

<b>Color Tag Examples</b>
<col fg="wfg" bg="wbg"> This is standard foreground / background text </col>
<col fg="empfg" bg="empbg"> This is emphasis foreground / background text </col>
<col fg="subfg" bg="subbg"> This is subdued foreground / background text </col>
<col fg="clfg" bg="clbg"> Test Text - Current Line</col>

<b>Other Tags Sets</b>
<col fg="normfg" bg="normbg"> Test Text - Normal (normfg / normbg) </col>
<col fg="warnfg" bg="warnbg"> Test Text - Warning (warnfg / warnbg) </col>
<col fg="errfg" bg="errbg"> Test Text - Error (errfg / errbg) </col>
<col fg="verbfg" bg="verbbg"> Test Text - Verbose (verbfg / verbbg) </col>

<b>Changed Text Tag Examples</b>
<col fg="changed" bg="wbg"> Test Text - Changed</col>

<b>Source Tags - using wbg background</b>
<col fg="srcnum" bg="wbg"> Test Text - srcnum  </col>
<col fg="srcchar" bg="wbg"> Test Text - srcchar  </col>
<col fg="srcstr" bg="wbg"> Test Text - srcstr  </col>
<col fg="srcid " bg="wbg"> Test Text - srcid   </col>
<col fg="srckw" bg="wbg"> Test Text - srckw </col>
<col fg="srcpair" bg="empbbg"> Test Text - srcpair </col>
<col fg="srccmnt" bg="wbg"> Test Text - srccmnt  </col>
<col fg="srcdrct" bg="wbg"> Test Text - srcdrct </col>
<col fg="srcspid" bg="wbg"> Test Text - srcspid </col>
<col fg="srcannot" bg="wbg"> Test Text - srcannot </col>

<b>Source Tags - using empbg background</b>
<col fg="srcnum" bg="empbg"> Test Text - srcnum  </col>
<col fg="srcchar" bg="empbg"> Test Text - srcchar  </col>
<col fg="srcstr" bg="empbg"> Test Text - srcstr  </col>
<col fg="srcid " bg="empbg"> Test Text - srcid   </col>
<col fg="srckw" bg="empbg"> Test Text - srckw </col>
<col fg="srcpair" bg="empbbg"> Test Text - srcpair </col>
<col fg="srccmnt" bg="empbg"> Test Text - srccmnt  </col>
<col fg="srcdrct" bg="empbg"> Test Text - srcdrct </col>
<col fg="srcspid" bg="empbg"> Test Text - srcspid </col>
<col fg="srcannot" bg="empbg"> Test Text - srcannot </col>

<b>Source Tags - using subbg background</b>
<col fg="srcnum" bg="subbg"> Test Text - srcnum  </col>
<col fg="srcchar" bg="subbg"> Test Text - srcchar  </col>
<col fg="srcstr" bg="subbg"> Test Text - srcstr  </col>
<col fg="srcid " bg="subbg"> Test Text - srcid   </col>
<col fg="srckw" bg="subbg"> Test Text - srckw </col>
<col fg="srcpair" bg="subbg"> Test Text - srcpair </col>
<col fg="srccmnt" bg="subbg"> Test Text - srccmnt  </col>
<col fg="srcdrct" bg="subbg"> Test Text - srcdrct </col>
<col fg="srcspid" bg="subbg"> Test Text - srcspid </col>
<col fg="srcannot" bg="subbg"> Test Text - srcannot </col>
```

DML Additions to the dbgeng Interface

The Debugger Engine and Extension APIs provide an interface to use the debugger engine to create custom applications. You can also write custom extensions that will run in WinDbg, KD, CDB, and NTSD. For more information see Writing DbgEng Extensions. This section describes the available DML enhancements to the debugger engine interfaces.

The dbgeng already has a set of text handling input methods and output interfaces, the use of DML only requires specification of the type of content carried in input and output text.

Providing DML Content to dbgeng

The output control flag DEBUG_OUTCTL_DML indicates that the text generated by a dbgeng method should be handled as DML content. If this flag is not given the text is treated as plain text context. DEBUG_OUTCTL_DML can be used with the following methods.

- **IDebugControl4::ControlledOutput**
- **IDebugControl4::ControlledOutputVaList**
- **IDebugControl4::ControlledOutputWide**
- **IDebugControl4::ControlledOutputVaListWide**

Text given must follow the DML rules for valid characters.

All output routines have been enhanced to allow a new format specifier %[h|w]Y{t}. This format specifier has a string pointer as an argument and indicates that the given text is plain text and should be converted to DML format during output processing. This gives callers a simple way of including plain text in DML content without having to pre-convert to DML format themselves. The h and w qualifiers indicate ANSI or Unicode text, as with %s.

The following table summarizes the use of the %Y format specifier.

%Y{t}	Quoted string. Will convert text to DML if the output format (first arg) is DML.
%Y{T}	Quoted string. Will always convert text to DML regardless of the output format.
%Y{s}	Unquoted string. Will convert text to DML if the output format (first arg) is DML.
%Y{S}	Unquoted string. Will always convert text to DML regardless of the output format.
%Y{as}	ULONG64. Adds either an empty string or 9 characters of spacing for padding the high 32-bit portion of debugger formatted pointer fields. The extra space outputs 9 spaces which includes the upper 8 zeros plus the ` character.
%Y{ps}	ULONG64. Extra space for padding debugger formatted pointer fields (includes the upper 8 zeros plus the ` character).
%Y{l}	ULONG64. Address as source line information.

This code snippet illustrates the use of the %Y format specifier.

```
HRESULT CALLBACK testout(_In_ PDEBUG_CLIENT pClient, _In_ PCWSTR /*pwszArgs*/)
{
    HRESULT hr = S_OK;

    ComPtr<IDebugControl4> spControl;
    IfFailedReturn(pClient->QueryInterface(IID_PPV_ARGS(&spControl)));

    spControl->ControlledOutputWide(DEBUG_OUTCTL_DML, DEBUG_OUTPUT_NORMAL, L"DML/NORMAL Y{t}: %Y{t}\n",
L"Hello <World>");
    spControl->ControlledOutputWide(DEBUG_OUTCTL_DML, DEBUG_OUTPUT_NORMAL, L"DML/NORMAL Y{T}: %Y{T}\n",
L"Hello <World>");
    spControl->ControlledOutputWide(DEBUG_OUTCTL_DML, DEBUG_OUTPUT_NORMAL, L"DML/NORMAL Y{s}: %Y{s}\n",
L"Hello <World>");
    spControl->ControlledOutputWide(DEBUG_OUTCTL_DML, DEBUG_OUTPUT_NORMAL, L"DML/NORMAL Y{S}: %Y{S}\n",
L"Hello <World>");

    spControl->ControlledOutputWide(0, DEBUG_OUTPUT_NORMAL, L"TEXT/NORMAL Y{t}: %Y{t}\n", L"Hello <World>");
    spControl->ControlledOutputWide(0, DEBUG_OUTPUT_NORMAL, L"TEXT/NORMAL Y{T}: %Y{T}\n", L"Hello <World>");
    spControl->ControlledOutputWide(0, DEBUG_OUTPUT_NORMAL, L"TEXT/NORMAL Y{s}: %Y{s}\n", L"Hello <World>");
    spControl->ControlledOutputWide(0, DEBUG_OUTPUT_NORMAL, L"TEXT/NORMAL Y{S}: %Y{S}\n", L"Hello <World>");

    spControl->ControlledOutputWide(DEBUG_OUTCTL_DML, DEBUG_OUTPUT_NORMAL, L"DML/NORMAL Y{a}: %Y{a}\n",
(ULONG64)0x00007ffa7da163c0);
    spControl->ControlledOutputWide(DEBUG_OUTCTL_DML, DEBUG_OUTPUT_NORMAL, L"DML/NORMAL Y{as} 64bit   :
'%Y{as}'\n", (ULONG64)0x00007ffa7da163c0);
    spControl->ControlledOutputWide(DEBUG_OUTCTL_DML, DEBUG_OUTPUT_NORMAL, L"DML/NORMAL Y{as} 32value :
'%Y{as}'\n", (ULONG64)0x1);

    spControl->ControlledOutputWide(DEBUG_OUTCTL_DML, DEBUG_OUTPUT_NORMAL, L"DML/NORMAL Y{ps} 64bit   :
'%Y{ps}'\n", (ULONG64)0x00007ffa7da163c0);
    spControl->ControlledOutputWide(DEBUG_OUTCTL_DML, DEBUG_OUTPUT_NORMAL, L"DML/NORMAL Y{ps} 32value :
'%Y{ps}'\n", (ULONG64)0x1);

    spControl->ControlledOutputWide(DEBUG_OUTCTL_DML, DEBUG_OUTPUT_NORMAL, L"DML/NORMAL Y{l}: %Y{l}\n",
(ULONG64)0x00007ffa7da163c0);

    return hr;

}
```

This sample code would generate the following output.

```
0:004> !testout
DML/NORMAL Y{t}: "Hello <World>"
DML/NORMAL Y{T}: "Hello <World>"
DML/NORMAL Y{s}: Hello <World>
DML/NORMAL Y{S}: Hello <World>
TEXT/NORMAL Y{t}: "Hello <World>"
TEXT/NORMAL Y{T}: "Hello &lt;World&gt;"
TEXT/NORMAL Y{s}: Hello <World>
TEXT/NORMAL Y{S}: Hello &lt;World&gt;
DML/NORMAL Y{a}: 00007ffa`7da163c0
DML/NORMAL Y{as} 64bit   : '         '
DML/NORMAL Y{as} 32value : '         '
DML/NORMAL Y{ps} 64bit   : '        '
DML/NORMAL Y{ps} 32value : '        '
DML/NORMAL Y{l}: [d:\th\minkernel\kernelbase\debug.c @ 443]
```

An additional control flag, DEBUG_OUTCTL_AMBIENT_DML, allows specification of DML context text without modifying any out output control attributes. DEBUG_OUTCTL_AMBIENT_TEXT has been added also as a more-descriptive alias for the previously-existing DEBUG_OUTCTL_AMBIENT. The output control flags are defined in dbgeng.h.

```
#define DEBUG_OUTCTL_DML               0x00000020

// Special values which mean leave the output settings
// unchanged.
#define DEBUG_OUTCTL_AMBIENT_DML       0xfffffffe
#define DEBUG_OUTCTL_AMBIENT_TEXT      0xffffffff

// Old ambient flag which maps to text.
#define DEBUG_OUTCTL_AMBIENT           DEBUG_OUTCTL_AMBIENT_TEXT
```

Providing DML Content From a Debuggee

The dbgeng has been enhanced to scan debuggee output for a special marker – – that indicates the remaining text in a piece of debuggee output should be treated as DML. The mode change only applies to a single piece of debuggee output, such as a single OutputDebugString string, and is not a global mode switch.

This example shows a mix of plain and DML output.

```
OutputDebugString("This is plain text\n<?dml?>This is <col fg=\"emphfg\">DML</col> text\n");
```

The output produced, will have a line of plain text followed by a line of DML where the acronym DML is displayed in a different color.

IDebugOutputCallbacks2

IDebugOutputCallbacks2 allows dbgeng interface clients to receive full DML content for presentation. IDebugOutputCallbacks2 is an extension of IDebugOutputCallbacks (not IDebugOutputCallbacksWide) so that it can be passed in to the existing SetOutputCallbacks method. The engine will do a QueryInterface for IDebugOutputCallbacks2 to see which interface the incoming output callback object supports. If the object supports IDebugOutputCallbacks2 all output will be sent through the extended IDebugOutputCallbacks2 methods; the basic IDebugOutputCallbacks::Output method will not be used.

The new methods are:

- IDebugOutputCallbacks2::GetInterestMask – Allows the callback object to describe which kinds of output notifications it wants to receive. The basic choice is between plain text content (DEBUG_OUTCBI_TEXT) and DML content (DEBUG_OUTCBI_DML). In addition, the callback object can also request notification of explicit flushes (DEBUG_OUTCBI_EXPLICIT_FLUSH).
- IDebugOutputCallbacks2::Output2 – All IDebugOutputCallbacks2 notifications come through Output2. The Which parameter indicates what kind of notification is coming in while the Flags, Arg and Text parameters carry the notification payload. Notifications include:
  - DEBUG_OUTCB_TEXT – Plain text output. Flags are from DEBUG_OUTCBF_*, Arg is the output mask and Text is the plain text. This will only be received if DEBUG_OUTCBI_TEXT was given in the interest mask.
  - DEBUG_OUTCB_DML – DML content output. Flags are from DEBUG_OUTCBF_*, Arg is the output mask and Text is the DML content. This will only be received if DEBUG_OUTCBI_DML was given in the interest mask.
  - DEBUG_OUTCB_EXPLICIT_FLUSH – A caller has called FlushCallbacks with no buffered text. Normally when buffered text is flushed the DEBUG_OUTCBF_COMBINED_EXPLICIT_FLUSH flag will be set, folding the two notifications into one. If no text is buffered a flush-only notification is sent.

The interest mask flags are defined in dbgeng.h as shown here.

```
// IDebugOutputCallbacks2 interest mask flags.
//
// Indicates that the callback wants notifications
// of all explicit flushes.
#define DEBUG_OUTCBI_EXPLICIT_FLUSH 0x00000001
// Indicates that the callback wants
// content in text form.
#define DEBUG_OUTCBI_TEXT           0x00000002
// Indicates that the callback wants
// content in markup form.
#define DEBUG_OUTCBI_DML            0x00000004

#define DEBUG_OUTCBI_ANY_FORMAT     0x00000006
```

Note that an output object can register for both text and DML content if it can handle them both. During output processing of the callback the engine will pick the format that reduces conversions, thus supporting both may reduce conversions in the engine. It is not necessary, though, and supporting only one format is the expected mode of operation.

Automatic Conversions

The dbgeng will automatically convert between plain text and DML as necessary. For example, if a caller sends DML content to the engine the engine will convert it to plain text for all output clients which only accept plain text. Alternately, the engine will convert plain text to DML for all output callbacks which only accept DML.

Related topics

Using Debugger Markup Language

JavaScript Debugger Scripting

9/12/2019 • 27 minutes to read • Edit Online

This topic describes how to use JavaScript to create scripts that understand debugger objects and extend and customize the capabilities of the debugger.

Overview of JavaScript Debugger Scripting

Script providers bridge a scripting language to the debugger's internal object model. The JavaScript debugger scripting provider, allows the for use of JavaScript with the debugger.

When a JavaScript is loaded via the .scriptload command, the root code of the script is executed, the names which are present in the script are bridged into the root namespace of the debugger (dx Debugger) and the script stays resident in memory until it is unloaded and all references to its objects are released. The script can provide new functions to the debugger's expression evaluator, modify the object model of the debugger, or can act as a visualizer in much the same way that a NatVis visualizer does.

This topic describes some of what you can do with JavaScript debugger scripting.

These two topics provide additional information about working with JavaScript in the debugger.

JavaScript Debugger Example Scripts

Native Objects in JavaScript Extensions

JavaScript Scripting Video

Defrag Tools #170 - Andy and Bill demonstrate JavaScript extensibility and scripting abilities in the debugger.

The Debugger JavaScript Provider

The JavaScript provider included with the debugger takes full advantage of the latest ECMAScript6 object and class enhancements. For more information, see ECMAScript 6 — New Features: Overview & Comparison.

JsProvider.dll

JsProvider.dll is the JavaScript provider that is loaded to support JavaScript Debugger Scripting.

Requirements

JavaScript Debugger Scripting is designed to work with all supported versions of Windows.

Loading the JavaScript Scripting Provider

Before using any of the .script commands, a scripting provider needs to be loaded using the **.load (Load Extension DLL)** command. To load the JavaScript provider, use the following command.

```
0:000> .load jsprovider.dll
```

Use the .scriptproviders command to confirm that the JavaScript provider is loaded.

```
0:000> .scriptproviders
Available Script Providers:
    NatVis (extension '.NatVis')
    JavaScript (extension '.js')
```

JavaScript Scripting Meta Commands

The following commands are available to work with JavaScript Debugger Scripting.

- **.scriptproviders (List Script Providers)**
- **.scriptload (Load Script)**
- **.scriptunload (Unload Script)**
- **.scriptrun (Run Script)**
- **.scriptlist (List Loaded Scripts)**

Requirements

Before using any of the .script commands, a scripting provider needs to be loaded using the **.load (Load Extension DLL)** command. To load the JavaScript provider, use the following command.

```
0:000> .load jsprovider.dll
```

.scriptproviders (List Script Providers)

The .scriptproviders command will list all the script languages which are presently understood by the debugger and the extension under which they are registered.

In the example below, the JavaScript and NatVis providers are loaded.

```
0:000> .scriptproviders
Available Script Providers:
    NatVis (extension '.NatVis')
    JavaScript (extension '.js')
```

Any file ending in ".NatVis" is understood as a NatVis script and any file ending in ".js" is understood as a JavaScript script. Either type of script can be loaded with the .scriptload command.

For more information, see **.scriptproviders (List Script Providers)**

.scriptload (Load Script)

The .scriptload command will load a script and execute the root code of a script and the *initializeScript* function. If there are any errors in the initial load and execution of the script, the errors will be displayed to console. The following command shows the successful load of TestScript.js.

```
0:000> .scriptload C:\WinDbg\Scripts\TestScript.js
JavaScript script successfully loaded from 'C:\WinDbg\Scripts\TestScript.js'
```

Any object model manipulations made by the script will stay in place until the script is subsequently unloaded or is run again with different content.

For more information, see **.scriptload (Load Script)**

.scriptrun

The .scriptrun command will load a script, execute the root code of the script, the *initializeScript* and the *invokeScript* function. If there are any errors in the initial load and execution of the script, the errors will be displayed to console.

```
0:000> .scriptrun C:\WinDbg\Scripts\helloWorld.js
JavaScript script successfully loaded from 'C:\WinDbg\Scripts\helloWorld.js'
Hello World!  We are in JavaScript!
```

Any debugger object model manipulations made by the script will stay in place until the script is subsequently unloaded or is run again with different content.

For more information, see **.scriptrun (Run Script)**.

.scriptunload (Unload Script)

The .scriptunload command unloads a loaded script and calls the *uninitializeScript* function. Use the following command syntax to unload a script

```
0:000:x86> .scriptunload C:\WinDbg\Scripts\TestScript.js
JavaScript script unloaded from 'C:\WinDbg\Scripts\TestScript.js'
```

For more information, see **.scriptunload (Unload Script)**.

.scriptlist (List Loaded Scripts)

The .scriptlist command will list any scripts which have been loaded via the .scriptload or the .scriptrun command. If the TestScript was successfully loaded using .scriptload, the .scriptlist command would display the name of the loaded script.

```
0:000> .scriptlist
Command Loaded Scripts:
    JavaScript script from 'C:\WinDbg\Scripts\TestScript.js'
```

For more information, see **.scriptlist (List Loaded Scripts)**.

Get Started with JavaScript Debugger Scripting

HelloWorld Example Script

This section describes how to create and execute a simple JavaScript debugger script that prints out, Hello World.

```
// WinDbg JavaScript sample
// Prints Hello World
function initializeScript()
{
    host.diagnostics.debugLog("***> Hello World! \n");
}
```

Use a text editor such as Notepad to create a text file named *HelloWorld.js* that contains the JavaScript code shown above.

Use the **.load (Load Extension DLL)** command to load the JavaScript provider.

```
0:000> .load jsprovider.dll
```

Use the .scriptload command to load and execute the script. Because we used the function name *initializeScript*, the code in the function is run when the script is loaded.

```
0:000> .scriptload c:\WinDbg\Scripts\HelloWorld.js
JavaScript script successfully loaded from 'c:\WinDbg\Scripts\HelloWorld.js'
***> Hello World!
```

After the script is loaded the additional functionality is available in the debugger. Use the **dx (Display NatVis Expression)** command to display *Debugger.State.Scripts* to see that our script is now resident.

```
0:000> dx Debugger.State.Scripts
Debugger.State.Scripts
    HelloWorld
```

In the next example, we will add and call a named function.

Adding Two Values Example Script

This section describes how to create and execute a simple JavaScript debugger script that adds takes input and adds two numbers.

This simple script provides a single function, addTwoValues.

```
// WinDbg JavaScript sample
// Adds two functions
function addTwoValues(a, b)
 {
     return a + b;
 }
```

Use a text editor such as Notepad to create a text file named *FirstSampleFunction.js*

Use the **.load (Load Extension DLL)** command to load the JavaScript provider.

```
0:000> .load jsprovider.dll
```

Use the .scriptload command to load the script.

```
0:000> .scriptload c:\WinDbg\Scripts\FirstSampleFunction.js
JavaScript script successfully loaded from 'c:\WinDbg\Scripts\FirstSampleFunction.js'
```

After the script is loaded the additional functionality is available in the debugger. Use the **dx (Display NatVis Expression)** command to display *Debugger.State.Scripts* to see that our script is now resident.

```
0:000> dx Debugger.State.Scripts
Debugger.State.Scripts
    FirstSampleFunction
```

We can click on the *FirstSampleFunction*, to see what functions it provides.

```
0:000> dx -r1 -v Debugger.State.Scripts.FirstSampleFunction.Contents
Debugger.State.Scripts.FirstSampleFunction.Contents                : [object Object]
    host             : [object Object]
    addTwoValues
 ...
```

To make the script a bit more convenient to work with, assign a variable in the debugger to hold the contents of the script using the dx command.

```
0:000> dx @$myScript = Debugger.State.Scripts.FirstSampleFunction.Contents
```

Use the dx expression evaluator to call the addTwoValues function.

```
0:000> dx @$myScript.addTwoValues(10, 41),d
@$myScript.addTwoValues(10, 41),d : 51
```

You can also use the *@$scriptContents* built in alias to work with the scripts. The *@$scriptContents* alias merges all of the .Content of all of the scripts that are loaded.

```
0:001> dx @$scriptContents.addTwoValues(10, 40),d
@$scriptContents.addTwoValues(10, 40),d : 50
```

When you are done working with the script use the .scriptunload command to unload the script.

```
0:000> .scriptunload c:\WinDbg\Scripts\FirstSampleFunction.js
JavaScript script successfully unloaded from 'c:\WinDbg\Scripts\FirstSampleFunction.js'
```

Debugger Command Automation

This section describes how to create and execute a simple JavaScript debugger script that automates the sending of the **u (Unassemble)** command. The sample also shows how to gather and display command output in a loop.

This script provides a single function, RunCommands().

```
// WinDbg JavaScript sample
// Shows how to call a debugger command and display results
"use strict";

function RunCommands()
{
var ctl = host.namespace.Debugger.Utility.Control;
var output = ctl.ExecuteCommand("u");
host.diagnostics.debugLog("***> Displaying command output \n");

for (var line of output)
   {
   host.diagnostics.debugLog("  ", line, "\n");
   }

host.diagnostics.debugLog("***> Exiting RunCommands Function \n");

}
```

Use a text editor such as Notepad to create a text file named *RunCommands.js*

Use the **.load (Load Extension DLL)** command to load the JavaScript provider.

```
0:000> .load jsprovider.dll
```

Use the .scriptload command to load the RunCommands script.

```
0:000> .scriptload c:\WinDbg\Scripts\RunCommands.js
JavaScript script successfully loaded from 'c:\WinDbg\Scripts\RunCommands.js'
```

After the script is loaded the additional functionality is available in the debugger. Use the **dx (Display NatVis Expression)** command to display *Debugger.State.Scripts.RunCommands* to see that our script is now resident.

```
0:000>dx -r3 Debugger.State.Scripts.RunCommands
Debugger.State.Scripts.RunCommands
    Contents         : [object Object]
        host             : [object Object]
            diagnostics      : [object Object]
            namespace
            currentSession   : Live user mode: <Local>
            currentProcess   : notepad.exe
            currentThread    : ntdll!DbgUiRemoteBreakin (00007ffd`87f2f440)
            memory           : [object Object]
```

Use the dx command to call the RunCommands function in the RunCommands script.

```
0:000> dx Debugger.State.Scripts.RunCommands.Contents.RunCommands()
  ***> Displaying command output
  ntdll!ExpInterlockedPopEntrySListEnd+0x17 [d:\rs1\minkernel\ntos\rtl\amd64\slist.asm @ 196]:
  00007ffd`87f06e67 cc              int     3
  00007ffd`87f06e68 cc              int     3
  00007ffd`87f06e69 0f1f8000000000  nop     dword ptr [rax]
  ntdll!RtlpInterlockedPushEntrySList [d:\rs1\minkernel\ntos\rtl\amd64\slist.asm @ 229]:
  00007ffd`87f06e70 0f0d09          prefetchw [rcx]
  00007ffd`87f06e73 53              push    rbx
  00007ffd`87f06e74 4c8bd1          mov     r10,rcx
  00007ffd`87f06e77 488bca          mov     rcx,rdx
  00007ffd`87f06e7a 4c8bda          mov     r11,rdx
***> Exiting RunCommands Function
```

Special JavaScript Debugger Functions

There are several special functions in a JavaScript script called by the script provider itself.

initializeScript

When a JavaScript script loads and is executed, it goes through a series of steps before the variables, functions, and other objects in the script affect the object model of the debugger.

- The script is loaded into memory and parsed.
- The root code in the script is executed.
- If the script has a method called initializeScript, that method is called.
- The return value from initializeScript is used to determine how to automatically modify the object model of the debugger.
- The names in the script are bridged to the debugger's namespace.

As mentioned, initializeScript will be called immediately after the root code of the script is executed. Its job is to return a JavaScript array of registration objects to the provider indicating how to modify the object model of the

debugger.

```
function initializeScript()
{
    // Add code here that you want to run every time the script is loaded.
    // We will just send a message to indicate that function was called.
    host.diagnostics.debugLog("***> initializeScript was called\n");
}
```

invokeScript

The invokeScript method is the primary script method and is called when .scriptload and .scriptrun are run.

```
function invokeScript()
{
    // Add code here that you want to run every time the script is executed.
    // We will just send a message to indicate that function was called.
    host.diagnostics.debugLog("***> invokeScript was called\n");
}
```

uninitializeScript

The uninitializeScript method is the behavioral opposite of initializeScript. It is called when a script is unlinked and is getting ready to unload. Its job is to undo any changes to the object model which the script made imperatively during execution and/or to destroy any objects which the script cached.

If a script neither makes imperative manipulations to the object model nor caches results, it does not need to have an uninitializeScript method. Any changes to the object model performed as indicated by the return value of initializeScript are undone automatically by the provider. Such changes do not require an explicit uninitializeScript method.

```
function uninitializeScript()
{
    // Add code here that you want to run every time the script is unloaded.
    // We will just send a message to indicate that function was called.
    host.diagnostics.debugLog("***> uninitialize was called\n");
}
```

Summary of Functions Called by Script Commands

This table summarizes which functions are called by the script commands

	.SCRIPTLOAD	.SCRIPTRUN (RUN SCRIPT)	.SCRIPTUNLOAD (UNLOAD SCRIPT)	
root	yes	yes		
initializeScript	yes	yes		
invokeScript		yes		
uninitializeScript			yes	

Use this sample code to see when each function is called as the script is loaded, executed and unloaded.

```
// Root of Script
host.diagnostics.debugLog("***>; Code at the very top (root) of the script is always run \n");


function initializeScript()
{
    // Add code here that you want to run every time the script is loaded.
    // We will just send a message to indicate that function was called.
    host.diagnostics.debugLog("***>; initializeScript was called \n");
}

function invokeScript()
{
    // Add code here that you want to run every time the script is executed.
    // We will just send a message to indicate that function was called.
    host.diagnostics.debugLog("***>; invokeScript was called \n");
}


function uninitializeScript()
{
    // Add code here that you want to run every time the script is unloaded.
    // We will just send a message to indicate that function was called.
    host.diagnostics.debugLog("***>; uninitialize was called\n");
}


function main()
{
    // main is just another function name in JavaScript
    // main is not called by .scriptload or .scriptrun
    host.diagnostics.debugLog("***>; main was called \n");
}
```

Creating a Debugger Visualizer in JavaScript

Custom visualization files allow you to group and organize data in a visualization structure that better reflects the data relationships and content. You can use the JavaScript debugger extensions to write debugger visualizers which act in a way very similar to NatVis. This is accomplished via authoring a JavaScript prototype object (or an ES6 class) which acts as the visualizer for a given data type. For more information about NatVis and the debugger see **dx (Display NatVis Expression)**.

Example class - Simple1DArray

Consider an example of a C++ class which represents a single dimensional array. This class has two members, m_size which is the overall size of the array, and m_pValues which is a pointer to a number of ints in memory equal to the m_size field.

```
class Simple1DArray
{
private:

    ULONG64 m_size;
    int *m_pValues;
};
```

We can use the dx command to look at the default data structure rendering.

```
0:000> dx g_array1D
g_array1D                 [Type: Simple1DArray]
    [+0x000] m_size           : 0x5 [Type: unsigned __int64]
    [+0x008] m_pValues        : 0x8be32449e0 : 0 [Type: int *]
```

JavaScript Visualizer

In order to visualize this type, we need to author a prototype (or ES6) class which has all the fields and properties we want the debugger to show. We also need to have the initializeScript method return an object which tells the JavaScript provider to link our prototype as a visualizer for the given type.

```
function initializeScript()
{
    //
    // Define a visualizer class for the object.
    //
    class myVisualizer
    {
        //
        // Create an ES6 generator function which yields back all the values in the array.
        //
        *[Symbol.iterator]()
        {
            var size = this.m_size;
            var ptr = this.m_pValues;
            for (var i = 0; i < size; ++i)
            {
                yield ptr.dereference();

                //
                // Note that the .add(1) method here is effectively doing pointer arithmetic on
                // the underlying pointer.  It is moving forward by the size of 1 object.
                //
                ptr = ptr.add(1);
            }
        }
    }

    return [new host.typeSignatureRegistration(myVisualizer, "Simple1DArray")];
}
```

Save the script in a file named arrayVisualizer.js.

Use the **.load (Load Extension DLL)** command to load the JavaScript provider.

```
0:000> .load C:\ScriptProviders\jsprovider.dll
```

Use .scriptload to load the array visualizer script.

```
0:000> .scriptload c:\WinDbg\Scripts\arrayVisualizer.js
JavaScript script successfully loaded from 'c:\WinDbg\Scripts\arrayVisualizer.js'
```

Now, when the dx command is used the script visualizer will display rows of array content.

```
0:000> dx g_array1D
g_array1D                 : [object Object] [Type: Simple1DArray]
    [<Raw View>]     [Type: Simple1DArray]
    [0x0]            : 0x0
    [0x1]            : 0x1
    [0x2]            : 0x2
    [0x3]            : 0x3
    [0x4]            : 0x4
```

In addition, this JavaScript visualization provides LINQ functionality, such as Select.

```
0:000> dx g_array1D.Select(n => n * 3),d
g_array1D.Select(n => n * 3),d
    [0]              : 0
    [1]              : 3
    [2]              : 6
    [3]              : 9
    [4]              : 12
```

What Affects the Visualization

A prototype or class which is made the visualizer for a native type through a return of a host.typeSignatureRegistration object from initializeScript will have all of the properties and methods within JavaScript added to the native type. In addition, the following semantics apply:

- Any name which does not start with two underscores (__) will be available in the visualization.
- Names which are part of standard JavaScript objects or are part of protocols which the JavaScript provider creates will not show up in the visualization.
- An object can be made iterable via the support of [Symbol.iterator].
- An object can be made indexable via the support of a custom protocol consisting of several functions: getDimensionality, getValueAt, and optionally setValueAt.

Native and JavaScript Object Bridge

The bridge between JavaScript and the object model of the debugger is two-way. Native objects can be passed into JavaScript and JavaScript objects can be passed into the Debugger's expression evaluator. As an example of this, consider the addition of the following method in our script:

```
function multiplyBySeven(val)
{
    return val * 7;
}
```

This method can now be utilized in the example LINQ query above. First we load the JavaScript visualization.

```
0:000> .scriptload c:\WinDbg\Scripts\arrayVisualizer2.js
JavaScript script successfully loaded from 'c:\WinDbg\Scripts\arrayVisualizer2.js'

0:000> dx @$myScript = Debugger.State.Scripts.arrayVisualizer2.Contents
```

Then we can use the multiplyBySeven function inline as shown below.

```
0:000> dx g_array1D.Select(@$myScript.multiplyBySeven),d
g_array1D.Select(@$myScript.multiplyBySeven),d
    [0]              : 0
    [1]              : 7
    [2]              : 14
    [3]              : 21
    [4]              : 28
```

Conditional Breakpoints with JavaScript

You can use JavaScript to do supplemental processing after a breakpoint is hit. For example, script can be used to examine other run time values and then determine if you want to automatically continue code execution or stop and do additional manual debugging.

For general information on working with breakpoints, see Methods of Controlling Breakpoints.

DebugHandler.js Example Breakpoint Processing Script

This example will evaluate notepad's open and save dialog: *notepad!ShowOpenSaveDialog*. This script will evaluate the pszCaption variable to determine if the current dialog is an "Open" dialog or if it is a "Save As" dialog. If it's an open dialog, code execution will continue. If it's a save as dialog, code execution will stop, and the debugger will break in.

```
 // Use JavaScript strict mode
"use strict";

// Define the invokeScript method to handle breakpoints

 function invokeScript()
 {
    var ctl = host.namespace.Debugger.Utility.Control;

    //Get the address of my string
    var address = host.evaluateExpression("pszCaption");

    // The open and save dialogs use the same function
    // When we hit the open dialog, continue.
    // When we hit the save dialog, break.
    if (host.memory.readWideString(address) == "Open") {
        // host.diagnostics.debugLog("We're opening, let's continue!\n");
        ctl.ExecuteCommand("gc");
    }
    else
    {
        //host.diagnostics.debugLog("We're saving, let's break!\n");
    }
  }
```

Use the **.load (Load Extension DLL)** command to load the JavaScript provider.

```
0:000> .load jsprovider.dll
```

This command sets a breakpoint on notepad!ShowOpenSaveDialog, and will run the script above whenever that breakpoint is hit.

```
bp notepad!ShowOpenSaveDialog ".scriptrun C:\\WinDbg\\Scripts\\DebugHandler.js"
```

Then when the File > Save option is selected in notepad, the script is run, the g command is not sent, and a

break in code execution occurs.

```
JavaScript script successfully loaded from 'C:\WinDbg\Scripts\DebugHandler.js'
notepad!ShowOpenSaveDialog:
00007ff6`f9761884 48895c2408      mov     qword ptr [rsp+8],rbx ss:000000db`d2a9f2f0=0000021985fe2060
```

Work With 64-Bit Values in JavaScript Extensions

This section describes how 64-bit values passed into a JavaScript debugger extension behave. This issue arises as JavaScript only has the ability to store numbers using 53-bits.

64-Bit and JavaScript 53-Bit Storage

Ordinal values passed into JavaScript are normally marshaled as JavaScript numbers. The problem with this is that JavaScript numbers are 64-bit double precision floating point values. Any ordinal over 53-bits would lose precision on entry into JavaScript. This presents an issue for 64-bit pointers and other 64-bit ordinal values which may have flags in the highest bytes. In order to deal with this, any 64-bit native value (whether from native code or the data model) which enters JavaScript enters as a library type -- not as a JavaScript number. This library type will round trip back to native code without losing numeric precision.

Auto-Conversion

The library type for 64-bit ordinal values supports the standard JavaScript valueOf conversion. If the object is used in a math operation or other construct which requires value conversion, it will automatically convert to a JavaScript number. If loss of precision were to occur (the value utilizes more than 53-bits of ordinal precision), the JavaScript provider will throw an exception.

Note that if you use bitwise operators in JavaScript, you are further limited to 32-bits of ordinal precision.

This sample code sums two numbers and will be used to test the conversion of 64 bit values.

```
function playWith64BitValues(a64, b64)
{
    // Sum two numbers to demonstrate 64-bit behavior.
    //
    // Imagine a64==100, b64==1000
    // The below would result in sum==1100 as a JavaScript number.  No exception is thrown.  The values
auto-convert.
    //
    // Imagine a64==2^56, b64=1
    // The below will **Throw an Exception**.  Conversion to numeric results in loss of precision!
    //
    var sum = a64 + b64;
    host.diagnostics.debugLog("Sum   >> ", sum, "\n");

}

function performOp64BitValues(a64, b64, op)
{
    //
    // Call a data model method passing 64-bit value.  There is no loss of precision here.  This round trips
perfectly.
    // For example:
    //  0:000> dx @$myScript.playWith64BitValues(0x4444444444444444ull, 0x3333333333333333ull, (x, y) => x +
y)
    //  @$myScript.playWith64BitValues(0x4444444444444444ull, 0x3333333333333333ull, (x, y) => x + y) :
0x7777777777777777
    //
    return op(a64, b64);
}
```

Use a text editor such as Notepad to create a text file named *PlayWith64BitValues.js*

Use the **.load (Load Extension DLL)** command to load the JavaScript provider.

```
0:000> .load jsprovider.dll
```

Use the .scriptload command to load the script.

```
0:000> .scriptload c:\WinDbg\Scripts\PlayWith64BitValues.js
JavaScript script successfully loaded from 'c:\WinDbg\Scripts\PlayWith64BitValues.js'
```

To make the script a bit more convenient to work with, assign a variable in the debugger to hold the contents of the script using the dx command.

```
0:000> dx @$myScript = Debugger.State.Scripts.PlayWith64BitValues.Contents
```

Use the dx expression evaluator to call the addTwoValues function.

First we will calculate the value of 2^53 =9007199254740992 (Hex 0x20000000000000).

First to test, we will use (2^53) - 2 and see that it returns the correct value for the sum.

```
0:000> dx @$myScript.playWith64BitValues(9007199254740990, 9007199254740990)
Sum   >> 18014398509481980
```

Then we will calculate (2^53) -1 =9007199254740991. This returns the error indicating that the conversion process will lose precision, so this is the largest value that can be used with the sum method in JavaScript code.

```
0:000> dx @$myScript.playWith64BitValues(9007199254740990, 9007199254740991)
Error: 64 bit value loses precision on conversion to number
```

Call a data model method passing 64-bit values. There is no loss of precision here.

```
0:001> dx @$myScript.performOp64BitValues( 0x7FFFFFFFFFFFFFF,  0x7FFFFFFFFFFFFFF, (x, y) => x + y)
@$myScript.performOp64BitValues( 0x7FFFFFFFFFFFFFF,  0x7FFFFFFFFFFFFFF, (x, y) => x + y) :
0xfffffffffffffffe
```

Comparison

The 64-bit library type is a JavaScript object and not a value type such as a JavaScript number. This has some implications for comparison operations. Normally, equality (==) on an object would indicate that operands refer to the same object and not the same value. The JavaScript provider mitigates this by tracking live references to 64-bit values and returning the same "immutable" object for non-collected 64-bit value. This means that for comparison, the following would occur.

```
// Comparison with 64 Bit Values

function comparisonWith64BitValues(a64, b64)
{
    //
    // No auto-conversion occurs here.  This is an *EFFECTIVE* value comparison.  This works with ordinals
with above 53-bits of precision.
    //
    var areEqual = (a64 == b64);
    host.diagnostics.debugLog("areEqual   >> ", areEqual, "\n");
    var areNotEqual = (a64 != b64);
    host.diagnostics.debugLog("areNotEqual   >> ", areNotEqual, "\n");

    //
    // Auto-conversion occurs here.  This will throw if a64 does not pack into a JavaScript number with no
loss of precision.
    //
    var isEqualTo42 = (a64 == 42);
    host.diagnostics.debugLog("isEqualTo42   >> ", isEqualTo42, "\n");
    var isLess = (a64 < b64);
    host.diagnostics.debugLog("isLess   >> ", isLess, "\n");
```

Use a text editor such as Notepad to create a text file named *ComparisonWith64BitValues.js*

Use the **.load (Load Extension DLL)** command to load the JavaScript provider.

```
0:000> .load jsprovider.dll
```

Use the .scriptload command to load the script.

```
0:000> .scriptload c:\WinDbg\Scripts\ComparisonWith64BitValues.js
JavaScript script successfully loaded from 'c:\WinDbg\Scripts\ComparisonWith64BitValues.js'
```

To make the script a bit more convenient to work with, assign a variable in the debugger to hold the contents of the script using the dx command.

```
0:000> dx @$myScript = Debugger.State.Scripts.comparisonWith64BitValues.Contents
```

First to test, we will use (2^53) - 2 and see that it returns the expected values.

```
0:001> dx @$myScript.comparisonWith64BitValues(9007199254740990, 9007199254740990)
areEqual   >> true
areNotEqual   >> false
isEqualTo42   >> false
isLess   >> false
```

We will also try the number 42 as the first value to validate the comparison operator is working as it should.

```
0:001> dx @$myScript.comparisonWith64BitValues(42, 9007199254740990)
areEqual   >> false
areNotEqual   >> true
isEqualTo42   >> true
isLess   >> true
```

Then we will calculate (2^53) -1 =9007199254740991. This value returns the error indicating that the conversion process will lose precision, so this is the largest value that can be used with the comparison

operators in JavaScript code.

```
0:000> dx @$myScript.playWith64BitValues(9007199254740990, 9007199254740991)
Error: 64 bit value loses precision on conversion to number
```

Maintaining Precision in Operations

In order to allow a debugger extension to maintain precision, a set of math functions are projected on top of the 64-bit library type. If the extension needs (or may possibly) need precision above 53-bits for incoming 64-bit values, the following methods should be utilized instead of relying on standard operators:

Method Name	Signature	Description
asNumber	.asNumber()	Converts the 64-bit value to a JavaScript number. If loss of precision occurs, **AN EXCEPTION IS THROWN**
convertToNumber	.convertToNumber()	Converts the 64-bit value to a JavaScript number. If loss of precision occurs, **NO EXCEPTION IS THROWN**
getLowPart	.getLowPart()	Converts the lower 32-bits of the 64-bit value to a JavaScript number
getHighPart	.getHighPart()	Converts the high 32-bits of the 64-bit value to a JavaScript number
add	.add(value)	Adds a value to the 64-bit value and returns the result
subtract	.subtract(value)	Subtracts a value from the 64-bit value and returns the result
multiply	.multiply(value)	Multiplies the 64-bit value by the supplied value and returns the result
divide	.divide(value)	Divides the 64-bit value by the supplied value and returns the result
bitwiseAnd	.bitwiseAnd(value)	Computes the bitwise and of the 64-bit value with the supplied value and returns the result
bitwiseOr	.bitwiseOr(value)	Computes the bitwise or of the 64-bit value with the supplied value and returns the result
bitwiseXor	.bitwiseXor(value)	Computes the bitwise xor of the 64-bit value with the supplied value and returns the result
bitwiseShiftLeft	.bitwiseShiftLeft(value)	Shifts the 64-bit value left by the given amount and returns the result

bitwiseShiftRight	.bitwiseShiftRight(value)	Shifts the 64-bit value right by the given amount and returns the result
toString	.toString([radix])	Converts the 64-bit value to a display string in the default radix (or the optionally supplied radix)

JavaScript Debugging

This section describes how to use the script debugging capabilities of the debugger. The debugger has integrated support for debugging JavaScript scripts using the .scriptdebug (Debug JavaScript) command.

> **NOTE**
> To use JavaScript Debugging with WinDbg Preview, run the debugger as Administrator.

Use this sample code to explore debugging a JavaScript. For this walkthrough, we will name it DebuggableSample.js and save it in the C:\MyScripts directory.

```
"use strict";

class myObj
{
    toString()
    {
        var x = undefined[42];
        host.diagnostics.debugLog("BOO!\n");
    }
}

class iterObj
{
    *[Symbol.iterator]()
    {
        throw new Error("Oopsies!");
    }
}

function foo()
{
    return new myObj();
}

function iter()
{
    return new iterObj();
}

function throwAndCatch()
{
    var outer = undefined;
    var someObj = {a : 99, b : {c : 32, d: "Hello World"} };
    var curProc = host.currentProcess;
    var curThread = host.currentThread;

    try
    {
        var x = undefined[42];
    } catch(e)
    {
        outer = e;
```

```
        outer = e;
    }

    host.diagnostics.debugLog("This is a fun test\n");
    host.diagnostics.debugLog("Of the script debugger\n");
    var foo = {a : 99, b : 72};
    host.diagnostics.debugLog("foo.a = ", foo.a, "\n");

    return outer;
}

function throwUnhandled()
{
    var proc = host.currentProcess;
    var thread = host.currentThread;
    host.diagnostics.debugLog("Hello...  About to throw an exception!\n");
    throw new Error("Oh me oh my!  This is an unhandled exception!\n");
    host.diagnostics.debugLog("Oh...  this will never be hit!\n");
    return proc;
}

function outer()
{
    host.diagnostics.debugLog("inside outer!\n");
    var foo = throwAndCatch();
    host.diagnostics.debugLog("Caught and returned!\n");
    return foo;
}

function outermost()
{
    var x = 99;
    var result = outer();
    var y = 32;
    host.diagnostics.debugLog("Test\n");
    return result;
}

function initializeScript()
{
    //
    // Return an array of registration objects to modify the object model of the debugger
    // See the following for more details:
    //
    //     https://aka.ms/JsDbgExt
    //
}
```

Load the sample script.

```
.scriptload C:\MyScripts\DebuggableSample.js
```

Start actively debugging the script using the **.scriptdebug** command.

```
0:000> .scriptdebug C:\MyScripts\DebuggableSample.js
>>> ****** DEBUGGER ENTRY DebuggableSample ******
           No active debug event!

>>> Debug [DebuggableSample <No Position>] >
```

Once you see the prompt >>> *Debug [DebuggableSample]* > and a request for input, you are inside the script debugger.

Use the **.help** command to display a list of commands in the JavaScript Debugging environment.

```
>>> Debug [DebuggableSample <No Position>] >.help
Script Debugger Commands (*NOTE* IDs are **PER SCRIPT**):
    ? ................................ Get help
    ? <expr>  ........................ Evaluate expression <expr> and display result
    ?? <expr>  ....................... Evaluate expression <expr> and display result
    |  ............................... List available scripts
    |<scriptid>s  ..................... Switch context to the given script
    bc <bpid>  ....................... Clear breakpoint by specified <bpid>
    bd <bpid>  ....................... Disable breakpoint by specified <bpid>
    be <bpid>  ....................... Enable breakpoint by specified <bpid>
    bl  .............................. List breakpoints
    bp <line>:<column>  .............. Set breakpoint at the specified line and column
    bp <function-name>  .............. Set breakpoint at the (global) function specified by the given name
    bpc  ............................. Set breakpoint at current location
    dv  .............................. Display local variables of current frame
    g  ............................... Continue script
    gu   ............................. Step out
    k  ............................... Get stack trace
    p  ............................... Step over
    q  ............................... Exit script debugger (resume execution)
    sx  .............................. Display available events/exceptions to break on
    sxe <event>  ..................... Enable break on <event>
    sxd <event>  ..................... Disable break on <event>
    t  ............................... Step in
    .attach <scriptId>  .............. Attach debugger to the script specified by <scriptId>
    .detach [<scriptId>]  ............ Detach debugger from the script specified by <scriptId>
    .frame <index>  .................. Switch to frame number <index>
    .f+  ............................. Switch to next stack frame
    .f-  ............................. Switch to previous stack frame
    .help  ........................... Get help
```

Use the **sx** script debugger command to see the list of events we can trap.

```
>>> Debug [DebuggableSample <No Position>] >sx
sx
    ab  [   inactive] .... Break on script abort
    eh  [   inactive] .... Break on any thrown exception
    en  [   inactive] .... Break on entry to the script
    uh  [     active] .... Break on unhandled exception
```

Use the **sxe** script debugger command to turn on break on entry so that the script will trap into the script debugger as soon as any code within it executes.

```
>>> Debug [DebuggableSample <No Position>] >sxe en
sxe en
Event filter 'en' is now active
```

Exit the script debugger and we'll make a function call into the script which will trap into the debugger.

```
>>> Debug [DebuggableSample <No Position>] >q
```

At this point, you are back in the normal debugger. Execute the following command to call the script.

```
dx @$scriptContents.outermost()
```

Now, you are back in the script debugger and broken in on the first line of the outermost JavaScript function.

```
>>> ****** SCRIPT BREAK DebuggableSample [BreakIn] ******
           Location: line = 73, column = 5
           Text: var x = 99

>>> Debug [DebuggableSample 73:5] >
```

In addition to seeing the break into the debugger, you get information on the line (73) and the column (5) where the break took place as well as the relevant snippet of source code: *var x = 99*.

Let's step a few times and get to another place in the script.

```
    p
    t
    p
    t
    p
    p
```

At this point, you should be broken into the throwAndCatch method on line 34.

```
...
>>> ****** SCRIPT BREAK DebuggableSample [Step Complete] ******
           Location: line = 34, column = 5
           Text: var curProc = host.currentProcess
```

You can verify this by executing a stack trace.

```
>>> Debug [DebuggableSample 34:5] >k
k
    ##  Function                        Pos    Source Snippet
-> [00] throwAndCatch                   034:05 (var curProc = host.currentProcess)
   [01] outer                           066:05 (var foo = throwAndCatch())
   [02] outermost                       074:05 (var result = outer())
```

From here, you can investigate the value of variables.

```
>>> Debug [DebuggableSample 34:5] >??someObj
??someObj
someObj          : {...}
    __proto__        : {...}
    a                : 0x63
    b                : {...}
>>> Debug [DebuggableSample 34:5] >??someObj.b
??someObj.b
someObj.b        : {...}
    __proto__        : {...}
    c                : 0x20
    d                : Hello World
```

Let's set a breakpoint on the current line of code and see what breakpoints are now set.

```
>>> Debug [DebuggableSample 34:5] >bpc
bpc
Breakpoint 1 set at 34:5
>>> Debug [DebuggableSample 34:5] >bl
bl
      Id State    Pos
       1 enabled  34:5
```

From here, we'll disable the entry (en) event using the **sxd** script debugger command.

```
>>> Debug [DebuggableSample 34:5] >sxd en
sxd en
Event filter 'en' is now inactive
```

And then just go and let the script continue to the end.

```
>>> Debug [DebuggableSample 34:5] >g
g
This is a fun test
Of the script debugger
foo.a = 99
Caught and returned!
Test
...
```

Execute the script method again and watch our breakpoint be hit.

```
0:000> dx @$scriptContents.outermost()
inside outer!
>>> ****** SCRIPT BREAK DebuggableSample [Breakpoint 1] ******
           Location: line = 34, column = 5
           Text: var curProc = host.currentProcess
```

Display the call stack.

```
>>> Debug [DebuggableSample 34:5] >k
k
    ##  Function                        Pos    Source Snippet
-> [00] throwAndCatch                   034:05 (var curProc = host.currentProcess)
   [01] outer                           066:05 (var foo = throwAndCatch())
   [02] outermost                       074:05 (var result = outer())
```

At this point, we want to stop debugging this script, so we detach from it.

```
>>> Debug [DebuggableSample 34:5] >.detach
.detach
Debugger has been detached from script!
```

And then type q to quit.

```
q
This is a fun test
Of the script debugger
foo.a = 99
Caught and returned!
Test
```

Executing the function again will no longer break into the debugger.

```
0:007> dx @$scriptContents.outermost()
inside outer!
This is a fun test
Of the script debugger
foo.a = 99
Caught and returned!
Test
```

JavaScript in VSCode - Adding IntelliSense

If you would like to work with the debugger data model objects in VSCode, you can use a definition file that is available in the Windows development kits. The IntelliSense definition file provides support for all of the host.* debugger object APIs. If you installed the kit in the default directory on a 64 bit PC, it is located here:

`C:\Program Files (x86)\Windows Kits\10\Debuggers\x64\winext\JsProvider.d.ts`

To use the IntelliSense definition file in VSCode:

1. Locate the definition file - JSProvider.d.ts
2. Copy the definition file to same folder as your script.
3. Add `/// <reference path="JSProvider.d.ts" />` to the top of your JavaScript script file.

With that reference in your JavaScript file, VS Code will automatically give you IntelliSense on the host APIs provided by JSProvider in addition to the structures in your script. For example, type "host." and you'll see IntelliSense for all the available debugger model APIs.

JavaScript Resources

The following are JavaScript resources that may be useful as you develop JavaScript debugging extensions.

- Writing JavaScript Code
- JScript Language Tour
- Mozilla JavaScript Reference
- WinJS: The Windows library for JavaScript
- ECMAScript 6 — New Features: Overview & Comparison

Related topics

JavaScript Debugger Example Scripts

Native Objects in JavaScript Extensions

JavaScript Debugger Example Scripts

4/19/2019 • 13 minutes to read • Edit Online

This topic provides the following user and kernel mode JavaScript code samples.

- Determining process architecture
- Data Filtering: Plug and Play Device Tree in KD (Kernel Mode)
- Extend Devices Specific To Multimedia (Kernel Mode)
- Adding Bus Information to _DEVICE_OBJECT (Kernel Mode)
- Find an Application Title (User Mode)

Microsoft GitHub Repo Example Scripts

The debugger team hosts a GitHub repo that contains example JavaScript scripts and extensions.

You can find it at - https://github.com/Microsoft/WinDbg-Samples

The readme file describes the current example code that is available.

Working with Samples

Use the general process to test any of the samples.

1. Determine if the sample JavaScript is intended for kernel or user mode debugging. Then either load an appropriate dump file or establish a live connection to a target system.

2. Use a text editor such as Notepad to create a text file named and save it with a .js file extension, such as *HelloWorld.js*

```
// WinDbg JavaScript sample
// Says Hello World!

// Code at root will be run with .scriptrun and .scriptload
host.diagnostics.debugLog("***> Hello World! \n");

function sayHi()
{
   //Say Hi
   host.diagnostics.debugLog("Hi from JavaScript! \n");
}
```

3. Use the **.load (Load Extension DLL)** command to load the JavaScript provider.

```
0:000> .load jsprovider.dll
```

4. Use the **.scriptrun (Run Script)**command to load and execute the script. The .scriptrun command will run code at the root/top and the code under the function names *initializeScript* and *invokeScript*.

```
0:000> .scriptrun c:\WinDbg\Scripts\HelloWorld.js
JavaScript script successfully loaded from 'c:\WinDbg\Scripts\HelloWorld.js'
***> Hello World!
```

5. If the script contains a uniquely named function, use the dx command to execute that function, that is located in Debugger.State.Scripts.*ScriptName*.Contents.*FunctionName*.

```
0:001> dx Debugger.State.Scripts.HelloWorld.Contents.sayHi()
Hi from JavaScript!
Debugger.State.Scripts.HelloWorld.Contents.sayHi()
```

Refer to JavaScript Debugger Scripting for additional information about working with JavaScript.

Determining process architecture

This JavaScript code adds a property called 'ProcessArchitecture' on to the debugger object model process object to indicate if the process is x86 or x64.

This script is intended to support kernel mode debugging.

```
"use strict";

class __CheckArchitecture
{
//
// Add a property called 'ProcessArchitecture' on process.
//
    get ProcessArchitecture()
    {
    var guestStates = this.Threads.Any(t=> (!(t.GuestState === undefined) && t.GuestState.Architecture
=="x86"));

        if(guestStates)
            return "x86";
        else
            return "x64";
    }
};

function initializeScript()
{
//
// Extends our notion of a process to place architecture information on it.
//
return [new host.namedModelParent(__CheckArchitecture, "Debugger.Models.Process")];
}
```

Either load a kernel dump file or establish a kernel mode connection to a target system. Then load the JavaScript provider and the sample script.

```
0: kd> !load jsprovider.dll
```

```
0: kd> .scriptload c:\WinDbg\Scripts\processarchitecture.js
JavaScript script successfully loaded from 'c:\WinDbg\Scripts\processarchitecture.js'
```

Use the dx command to display the process architecture of the current process.

```
2: kd> dx @$curprocess
@$curprocess                 : System [Switch To]
    KernelObject     [Type: _EPROCESS]
    Name             : System
    Id               : 0x4
    Handle           : 0xf0f0f0f0
    Threads
    Modules
    Environment
    Devices
    Io
    ProcessArchitecture : x64
```

Note that this sample code may not always be able to determine the architecture correctly. For example in certain cases working with dump files when you are using the 32-bit debugger.

Data Filtering: Plug and Play Device Tree in KD (Kernel Mode)

This sample code filters the device node tree to display just devices that contain a path of PCI that are started.

This script is intended to support kernel mode debugging.

You can use the !devnode 0 1 command to display information about the device tree. For more information, see **!devnode**.

```
// PlugAndPlayDeviceTree.js
// An ES6 generator function which recursively filters the device tree looking for PCI devices in the started
state.
//
function *filterDevices(deviceNode)
{
    //
    // If the device instance path has "PCI" in it and is started (state == 776), yield it from the
generator.
    //
    if (deviceNode.InstancePath.indexOf("PCI") != -1 && deviceNode.State == 776)
    {
        yield deviceNode;
    }

    //
    // Recursively invoke the generator for all children of the device node.
    //
    for (var childNode of deviceNode.Children)
    {
        yield* filterDevices(childNode);
    }
}

//
// A function which finds the device tree of the first session in the debugger and passes it to our filter
function.
//
function filterAllDevices()
{
    return filterDevices(host.namespace.Debugger.Sessions.First().Devices.DeviceTree.First());
}
```

Either load a kernel dump file or establish a kernel mode connection to a target system.

```
0: kd> !load jsprovider.dll
```

```
0: kd> .scriptload c:\WinDbg\Scripts\deviceFilter.js
JavaScript script successfully loaded from 'c:\WinDbg\Scripts\deviceFilter.js'
```

Call the filterAllDevices() function.

```
0: kd> dx Debugger.State.Scripts.PlugAndPlayDeviceTree.Contents.filterAllDevices()
Debugger.State.Scripts.PlugAndPlayDeviceTree.Contents.filterAllDevices()                 : [object Generator]
    [0x0]            : PCI\VEN_8086&DEV_D131&SUBSYS_304A103C&REV_11\3&21436425&0&00
    [0x1]            : PCI\VEN_8086&DEV_D138&SUBSYS_304A103C&REV_11\3&21436425&0&18 (pci)
    [0x2]            : PCI\VEN_10DE&DEV_06FD&SUBSYS_062E10DE&REV_A1\4&324c21a&0&0018 (nvlddmkm)
    [0x3]            : PCI\VEN_8086&DEV_3B64&SUBSYS_304A103C&REV_06\3&21436425&0&B0 (HECIx64)
    [0x4]            : PCI\VEN_8086&DEV_3B3C&SUBSYS_304A103C&REV_05\3&21436425&0&D0 (usbehci)
    [0x5]            : PCI\VEN_8086&DEV_3B56&SUBSYS_304A103C&REV_05\3&21436425&0&D8 (HDAudBus)
...
```

Each of these objects presented above, automatically supports DML, and can be clicked through just as with any other dx query.

Alternatively to using this script, it is possible to use a LINQ query to accomplish a similar result.

```
0: kd> dx @$cursession.Devices.DeviceTree.Flatten(n => n.Children).Where(n => n.InstancePath.Contains("PCI")
&& n.State == 776)
@$cursession.Devices.DeviceTree.Flatten(n => n.Children).Where(n => n.InstancePath.Contains("PCI") && n.State
== 776)
    [0x0]            : PCI\VEN_8086&DEV_D131&SUBSYS_304A103C&REV_11\3&21436425&0&00
    [0x1]            : PCI\VEN_8086&DEV_D138&SUBSYS_304A103C&REV_11\3&21436425&0&18 (pci)
    [0x2]            : PCI\VEN_10DE&DEV_06FD&SUBSYS_062E10DE&REV_A1\4&324c21a&0&0018 (nvlddmkm)
    [0x3]            : PCI\VEN_8086&DEV_3B64&SUBSYS_304A103C&REV_06\3&21436425&0&B0 (HECIx64)
    [0x4]            : PCI\VEN_8086&DEV_3B3C&SUBSYS_304A103C&REV_05\3&21436425&0&D0 (usbehci)
    [0x5]            : PCI\VEN_8086&DEV_3B56&SUBSYS_304A103C&REV_05\3&21436425&0&D8 (HDAudBus)
...
```

Extend Devices Specific To Multimedia (Kernel Mode)

This larger JavaScript example extends a kernel _DEVICE_OBJECT for information specific to multimedia and adds StreamingDevices to a debugger session.

This script is intended to support kernel mode debugging.

Note that the choice to extend Session with StreamingDevices is done for example purposes only. This should be either left to _DEVICE_OBJECT only or deeper inside a namespace under the existing .Devices.* hierarchy.

```
// StreamingFinder.js
// Extends a kernel _DEVICE_OBJECT for information specific to multimedia
// and adds StreamingDevices to a debugger session.
//

"use strict";

function initializeScript()
{
    // isStreamingDeviceObject:
    //
    // Returns whether devObj (as a _DEVICE_OBJECT -- not a pointer) looks like it is a device
    // object for a streaming device.
    //
    function isStreamingDeviceObject(devObj)
    {
        try
        {
```

```
            var devExt = devObj.DeviceExtension;
            var possibleStreamingExtPtrPtr = host.createPointerObject(devExt.address, "ks.sys",
"_KSIDEVICE_HEADER **", devObj);
            var possibleStreamingExt = possibleStreamingExtPtrPtr.dereference().dereference();
            var baseDevice = possibleStreamingExt.BaseDevice;

            if (devObj.targetLocation == baseDevice.dereference().targetLocation)
            {
                return true;
            }
        }
        //
        // The above code expects to fail (walking into invalid or paged out memory) often.  A failure to
read the memory
        // of the target process will result in an exception.  Catch such exception and indicate that the
object does not
        // match the profile of a "streaming device object".
        //
        catch(exc)
        {
        }

        return false;
    }

    // findStreamingFDO(pdo):
    //
    // From a physical device object, walks up the device stack and attempts to find a device which
    // looks like a streaming device (and is thus assumed to be the FDO).  pdo is a pointer to
    // the _DEVICE_OBJECT for the pdo.
    //
    function findStreamingFDO(pdo)
    {
        for (var device of pdo.UpperDevices)
        {
            if (isStreamingDeviceObject(device.dereference()))
            {
                return device;
            }
        }

        return null;
    }

    // streamingDeviceList:
    //
    // A class which enumerates all streaming devices on the system.
    //
    class streamingDeviceList
    {
        constructor(session)
        {
            this.__session = session;
        }

        *[Symbol.iterator]()
        {
            //
            // Get the list of all PDOs from PNP using LINQ:
            //
            var allPDOs = this.__session.Devices.DeviceTree.Flatten(function(dev) { return dev.Children; })
                                                           .Select (function(dev) { return
dev.PhysicalDeviceObject; });

            //
            // Walk the stack up each PDO to find the functional device which looks like a KS device.  This
is
            // a very simple heuristic test: the back pointer to the device in the KS device extension is
            // accurate.  Make sure this is wrapped in a try/catch to avoid invalid memory reads causing
```

```
            // us to bail out.
            //
            // Don't even bother checking the PDO.
            //
            for (var pdo of allPDOs)
            {
                var fdo = findStreamingFDO(pdo);
                if (fdo != null)
                {
                    yield fdo;
                }
            }
        }
    }

    // streamingDeviceExtension:
    //
    // An object which will extend "session" and include the list of streaming devices.
    //
    class streamingDeviceExtension
    {
        get StreamingDevices()
        {
            return new streamingDeviceList(this);
        }
    };

    // createEntryList:
    //
    // An abstraction over the create entry list within a streaming device.
    //
    class createEntryList
    {
        constructor(ksHeader)
        {
            this.__ksHeader = ksHeader;
        }

        *[Symbol.iterator]()
        {
            for (var entry of
host.namespace.Debugger.Utility.Collections.FromListEntry(this.__ksHeader.ChildCreateHandlerList,
"ks!KSICREATE_ENTRY", "ListEntry"))
            {
                                                                if (!entry.CreateItem.Create.isNull)
                {
                    yield entry;
                }
            }
        }
    }

    // streamingInformation:
    //
    // Represents the streaming state of a device.
    //
    class streamingInformation
    {
        constructor(fdo)
        {
            this.__fdo = fdo;

            var devExt = fdo.DeviceExtension;
            var streamingExtPtrPtr = host.createPointerObject(devExt.address, "ks.sys", "_KSIDEVICE_HEADER
**", fdo);
            this.__ksHeader = streamingExtPtrPtr.dereference().dereference();
        }

        get CreateEntries()
```

```
        {
            return new createEntryList(this.__ksHeader);
        }
    }

    // createEntryVisualizer:
    //
    // A visualizer for KSICREATE_ENTRY
    //
    class createEntryVisualizer
    {
        toString()
        {
            return this.CreateItem.ObjectClass.toString();
        }

        get CreateContext()
        {
            //
            // This is probably not entirely accurate.  The context is not *REQUIRED* to be an IUnknown.
            // More analysis should probably be performed.
            //
            return host.createTypedObject(this.CreateItem.Context.address, "ks.sys", "IUnknown", this);
        }
    }

    // deviceExtension:
    //
    // Extends our notion of a device in the device tree to place streaming information on
    // top of it.
    //
    class deviceExtension
    {
        get StreamingState()
        {
            if (isStreamingDeviceObject(this))
            {
                return new streamingInformation(this);
            }

            //
            // If we cannot find a streaming FDO, returning undefined will indicate that there is no value
            // to the property.
            //
            return undefined;
        }
    }

    return [new host.namedModelParent(streamingDeviceExtension, "Debugger.Models.Session"),
            new host.typeSignatureExtension(deviceExtension, "_DEVICE_OBJECT"),
            new host.typeSignatureRegistration(createEntryVisualizer, "KSICREATE_ENTRY")]
}
```

First load the script provider as described previously. Then load the script.

```
0: kd> .scriptload c:\WinDbg\Scripts\StreamingFinder.js
JavaScript script successfully loaded from 'c:\WinDbg\Scripts\StreamingFinder.js'
```

Then use the dx command to access the new StreamingDevices capabilities that the script provides.

```
0: kd> dx -r3 @$cursession.StreamingDevices.Select(d => d->StreamingState.CreateEntries)
@$cursession.StreamingDevices.Select(d => d->StreamingState.CreateEntries)
    [0x0]            : [object Object]
        [0x0]            : "e0HDMIOutTopo" [Type: KSICREATE_ENTRY]
            [<Raw View>]     [Type: KSICREATE_ENTRY]
            CreateContext    [Type: CPortTopology]
    [0x1]            : [object Object]
        [0x0]            : "AnalogDigitalCaptureTopo" [Type: KSICREATE_ENTRY]
            [<Raw View>]     [Type: KSICREATE_ENTRY]
            CreateContext    [Type: CPortTopology]
        [0x1]            : "AnalogDigitalCapture1Topo" [Type: KSICREATE_ENTRY]
            [<Raw View>]     [Type: KSICREATE_ENTRY]
            CreateContext    [Type: CPortTopology]
        [0x2]            : "AnalogDigitalCapture2Topo" [Type: KSICREATE_ENTRY]
            [<Raw View>]     [Type: KSICREATE_ENTRY]
            CreateContext    [Type: CPortTopology]
        [0x3]            : "AnalogDigitalCapture2Wave" [Type: KSICREATE_ENTRY]
            [<Raw View>]     [Type: KSICREATE_ENTRY]
            CreateContext    [Type: CPortWaveRT]
        [0x4]            : "HeadphoneTopo" [Type: KSICREATE_ENTRY]
            [<Raw View>]     [Type: KSICREATE_ENTRY]
            CreateContext    [Type: CPortTopology]
        [0x5]            : "RearLineOutTopo" [Type: KSICREATE_ENTRY]
            [<Raw View>]     [Type: KSICREATE_ENTRY]
            CreateContext    [Type: CPortTopology]
        [0x6]            : "RearLineOutWave" [Type: KSICREATE_ENTRY]
            [<Raw View>]     [Type: KSICREATE_ENTRY]
            CreateContext    [Type: CPortWaveRT]
    [0x2]            : [object Object]
        [0x0]            : "GLOBAL" [Type: KSICREATE_ENTRY]
            [<Raw View>]     [Type: KSICREATE_ENTRY]
            CreateContext    [Type: IUnknown]
```

Adding Bus Information to _DEVICE_OBJECT (Kernel Mode)

This script extends the visualization of _DEVICE_OBJECT to add a BusInformation field which has PCI specific information underneath it. The manner and namespacing of this sample are still being discussed. It should be considered a sample of the capabilities of the JavaScript provider.

This script is intended to support kernel mode debugging.

```
"use strict";

/***********************************************
DeviceExtensionInformation.js:

An example which extends _DEVICE_OBJECT to add bus specific information
to each device object.

NOTE: The means of conditionally adding and the style of namespacing this
are still being discussed.  This currently serves as an example of the capability
of JavaScript extensions.

***********************************************/

function initializeScript()
{
    // __getStackPDO():
    //
    // Returns the physical device object of the device stack whose device object
    // is passed in as 'devObj'.
    //
    function __getStackPDO(devObj)
    {
```

```
        var curDevice = devObj;
        var nextDevice = curDevice.DeviceObjectExtension.AttachedTo;
        while (!nextDevice.isNull)
        {
            curDevice = nextDevice;
            nextDevice = curDevice.DeviceObjectExtension.AttachedTo;
        }
        return curDevice;
    }

    // pciInformation:
    //
    // Class which abstracts our particular representation of PCI information for a PCI device or bus
    // based on a _PCI_DEVICE structure or a _PCI_BUS structure.
    //
    class pciInformation
    {
        constructor(pciDev)
        {
            this.__pciDev = pciDev;
            this.__pciPDO = __getStackPDO(this.__pciDev);
            this.__isBridge = (this.__pciDev.address != this.__pciPDO.address);

            if (this.__isBridge)
            {
                this.__deviceExtension = host.createTypedObject(this.__pciPDO.DeviceExtension.address,
"pci.sys", "_PCI_DEVICE", this.__pciPDO);
                this.__busExtension = host.createTypedObject(this.__pciDev.DeviceExtension.address,
"pci.sys", "_PCI_BUS", this.__pciPDO);

                this.__hasDevice = (this.__deviceExtension.Signature == 0x44696350); /* 'PciD' */
                this.__hasBus = (this.__busExtension.Signature == 0x42696350); /* 'PciB' */

                if (!this.__hasDevice && !this.__hasBus)
                {
                    throw new Error("Unrecognized PCI device extension");
                }
            }
            else
            {
                this.__deviceExtension = host.createTypedObject(this.__pciPDO.DeviceExtension.address,
"pci.sys", "_PCI_DEVICE", this.__pciPDO);

                this.__hasDevice = (this.__deviceExtension.Signature == 0x44696350); /* 'PciD' */
                this.__hasBus = false;

                if (!this.__hasDevice)
                {
                    throw new Error("Unrecognized PCI device extension");
                }
            }
        }

        toString()
        {

            if (this.__hasBus && this.__hasDevice)
            {
                return "Bus: " + this.__busExtension.toString() + " Device: " +
this.__deviceExtension.toString();
            }
            else if (this.__hasBus)
            {
                return this.__busExtension.toString();
            }
            else
            {
                return this.__deviceExtension.toString();
            }
```

```
        }

        get Device()
        {
            if (this.__hasDevice)
            {
                // NatVis supplies the visualization for _PCI_DEVICE
                return this.__deviceExtension;
            }

            return undefined;
        }

        get Bus()
        {
            if (this.__hasBus)
            {
                // NatVis supplies the visualization for _PCI_BUS
                return this.__busExtension;
            }

            return undefined;
        }
    }

    // busInformation:
    //
    // Our class which does analysis of what bus a particular device is on and places information about
    // about that bus within the _DEVICE_OBJECT visualization.
    //
    class busInformation
    {
        constructor(devObj)
        {
            this.__devObj = devObj;
        }

        get PCI()
        {
            //
            // Check the current device object.  This may be a PCI bridge
            // in which both the FDO and PDO are relevant (one for the bus information
            // and one for the bridge device information).
            //
            var curName = this.__devObj.DriverObject.DriverName.toString();
            if (curName.includes("\\Driver\\pci"))
            {
                return new pciInformation(this.__devObj);
            }

            var stackPDO = __getStackPDO(this.__devObj);
            var pdoName = stackPDO.DriverObject.DriverName.toString();
            if (pdoName.includes("\\Driver\\pci"))
            {
                return new pciInformation(stackPDO);
            }

            return undefined;
        }
    }

    // busInformationExtension:
    //
    // An extension placed on top of _DEVICE_OBJECT in order to provide bus analysis and bus specific
    // information to the device.
    //
    class busInformationExtension
    {
        get BusInformation()
```

```
        {
            return new busInformation(this);
        }
    }

    return [new host.typeSignatureExtension(busInformationExtension, "_DEVICE_OBJECT")];
}
```

First load the script provider as described previously. Then load the script.

```
0: kd> .scriptload c:\WinDbg\Scripts\DeviceExtensionInformation.js
JavaScript script successfully loaded from 'c:\WinDbg\Scripts\DeviceExtensionInformation.js'
```

We need to locate the address of the device object we are interested in. In this example, we will examine the audio HDAudBus driver.

```
0: kd>  !drvobj HDAudBus
Driver object (ffffb60757a4ae60) is for:
 \Driver\HDAudBus
Driver Extension List: (id , addr)
(fffff8050a9eb290 ffffb60758413180)
Device Object list:
ffffb60758e21810  ffffb60757a67c60
```

After the script is loaded use the dx command to display bus information for device objects.

```
0: kd> dx -r1 (*((ntkrnlmp!_DEVICE_OBJECT *)0xffffe00001b567c0))
(*((ntkrnlmp!_DEVICE_OBJECT *)0xffffe00001b567c0))                : Device for "\Driver\HDAudBus" [Type:
_DEVICE_OBJECT]
    [<Raw View>]     [Type: _DEVICE_OBJECT]
    Flags            : 0x2004
    UpperDevices     : None
    LowerDevices     : Immediately below is Device for "\Driver\ACPI" [at 0xffffe000003d9820]
    Driver           : 0xffffe00001ccd060 : Driver "\Driver\HDAudBus" [Type: _DRIVER_OBJECT *]
    BusInformation   : [object Object]

0: kd> dx -r1 (*((ntkrnlmp!_DEVICE_OBJECT *)0xffffe00001b567c0)).@"BusInformation"
(*((ntkrnlmp!_DEVICE_OBJECT *)0xffffe00001b567c0)).@"BusInformation"                : [object Object]
    PCI              :  (d=0x14 f=0x2) Vendor=0x1022 Device=0x780d Multimedia Device / Unknown Sub Class

0: kd> dx -r1 (*((ntkrnlmp!_DEVICE_OBJECT *)0xffffe00001b567c0)).@"BusInformation".@"PCI"
(*((ntkrnlmp!_DEVICE_OBJECT *)0xffffe00001b567c0)).@"BusInformation".@"PCI"                :  (d=0x14 f=0x2)
Vendor=0x1022 Device=0x780d Multimedia Device / Unknown Sub Class
    Device           :  (d=0x14 f=0x2) Vendor=0x1022 Device=0x780d Multimedia Device / Unknown Sub Class
[Type: _PCI_DEVICE]

0: kd> dx -r1 (*((pci!_PCI_DEVICE *)0xffffe000003fe1b0))
(*((pci!_PCI_DEVICE *)0xffffe000003fe1b0))                :  (d=0x14 f=0x2) Vendor=0x1022 Device=0x780d
Multimedia Device / Unknown Sub Class [Type: _PCI_DEVICE]
    [<Raw View>]     [Type: _PCI_DEVICE]
    Device           : 0xffffe000003fe060 : Device for "\Driver\pci" [Type: _DEVICE_OBJECT *]
    Requirements
    Resources

0: kd> dx -r1 (*((pci!_PCI_DEVICE *)0xffffe000003fe1b0)).@"Resources"
(*((pci!_PCI_DEVICE *)0xffffe000003fe1b0)).@"Resources"
    BaseAddressRegisters
    Interrupt        : Line Based -- Interrupt Line = 0x10 [Type: _PCI_DEVICE_INTERRUPT_RESOURCE]

0: kd> dx -r1 (*((pci!_PCI_DEVICE *)0xffffe000003fe1b0)).@"Resources".@"BaseAddressRegisters"
(*((pci!_PCI_DEVICE *)0xffffe000003fe1b0)).@"Resources".@"BaseAddressRegisters"
    [0x0]            : Memory Resource: 0xf0340000 of length 0x4000 [Type: _CM_PARTIAL_RESOURCE_DESCRIPTOR]
```

Find an Application Title (User Mode)

This example iterates through all the threads in the debugger's current process, finds a frame which includes *__mainCRTStartup* and then returns the string from the StartupInfo.lpTitle within the CRT's startup. This script shows examples of iteration, string manipulation, and LINQ queries within JavaScript.

This script is intended to support user mode debugging.

```
// TitleFinder.js
// A function which uses just JavaScript concepts to find the title of an application
//
function findTitle()
{
    var curProcess = host.currentProcess;
    for (var thread of curProcess.Threads)
    {
        for (var frame of thread.Stack.Frames)
        {
            if (frame.toString().includes("__mainCRTStartup"))
            {
                var locals = frame.LocalVariables;

                //
                // locals.StartupInfo.lpTitle is just an unsigned short *.  We need to actually call an API
to
                // read the UTF-16 string in the target address space.  This would be true even if this were
                // a char* or wchar_t*.
                //
                return host.memory.readWideString(locals.StartupInfo.lpTitle);
            }
        }
    }
}

//
// A function which uses both JavaScript and integrated LINQ concepts to do the same.
//
function findTitleWithLINQ()
{
    var isMainFrame = function(frame) { return frame.toString().includes("__mainCRTStartup"); };
    var isMainThread = function(thread) { return thread.Stack.Frames.Any(isMainFrame); };

    var curProcess = host.currentProcess;
    var mainThread = curProcess.Threads.Where(isMainThread).First();
    var mainFrame = mainThread.Stack.Frames.Where(isMainFrame).First();

    var locals = mainFrame.LocalVariables;

    //
    // locals.StartupInfo.lpTitle is just an unsigned short *.  We need to actually call an API to
    // read the UTF-16 string in the target address space.  This would be true even if this were
    // a char* or wchar_t*.
    //
    return host.memory.readWideString(locals.StartupInfo.lpTitle);
}
```

```
0: kd> .scriptload c:\WinDbg\Scripts\TitleFinder.js
JavaScript script successfully loaded from 'c:\WinDbg\Scripts\TitleFinder.js'
```

Calling the findTitle() function returns notepad.exe

```
0:000> dx Debugger.State.Scripts.TitleFinder.Contents.findTitle()
Debugger.State.Scripts.TitleFinder.Contents.findTitle() : C:\Windows\System32\notepad.exe
```

Calling the LINQ version, findTitleWithLINQ() also returns notepad.exe

```
0:000> dx Debugger.State.Scripts.TitleFinder.Contents.findTitleWithLINQ()
Debugger.State.Scripts.titleFinder.Contents.findTitleWithLINQ() : C:\Windows\System32\notepad.exe
```

Related topics

JavaScript Debugger Scripting

Native Debugger Objects in JavaScript Extensions

9/12/2019 • 10 minutes to read • Edit Online

Native debugger objects represent various constructs and behaviors of the debugger environment. The objects can be passed into (or acquired in) JavaScript extensions to manipulate the state of the debugger.

Example debugger objects include the following.

- Session
- Threads / Thread
- Processes / Process
- Stack Frames / Stack Frame
- Local Variables
- Modules / Module
- Utility
- State
- Settings

For example the host.namespace.Debugger.Utility.Control.ExecuteCommand object can be used to send the u command to the debugger with following two lines of JavaScript code.

```
var ctl = host.namespace.Debugger.Utility.Control;
var outputLines = ctl.ExecuteCommand("u");
```

This topic describes how to work with common objects and provides reference information on their attributes and behaviors.

For general information about working with JavaScript, see JavaScript Debugger Scripting. For JavaScript examples that use the debugger objects, see JavaScript Debugger Example Scripts. For information about working with the settings objects, see **.settings (Set Debug Settings)**.

To explore the objects available in a debugger session, use the **dx (Display NatVis Expression)** command. For example, you can display some of the top level debugger objects with this dx command.

```
0: kd> dx -r2 Debugger
Debugger
    Sessions         : [object Object]
        [0x0]            : Remote KD: KdSrv:Server=@{<Local>},Trans=@{NET:Port=50000,Key=1.2.3.4,Target}
    Settings
        Debug
        Display
        EngineInitialization
        Extensions
        Input
        Sources
        Symbols
        AutoSaveSettings : false
    State
        DebuggerVariables
        PseudoRegisters
        Scripts
        UserVariables
    Utility
        Collections
        Control
        Objects
```

All of the items listed above are clickable DML and can be recursed further down to view the debugger object structure.

Extending the Debugger via the Data Model

The debugger data model allows for the creation of an interface to information about applications and drivers in Windows that has the following attributes.

- Is discoverable and organized- a logically structured name space can be queried using the dx command.
- Can be queried using LINQ- This allows for extraction and sorting of data using a standard query language.
- Can be logically and consistently extended - Extensible using techniques described in this topic with debugger scripting providers such as Natvis and JavaScript.

Extending a Debugger Object in JavaScript

In addition to being able to create a visualizer in JavaScript, script extensions can also modify the core concepts of the debugger - sessions, processes, threads, stacks, stack frames, local variables - and even publish themselves as extension points that other extensions can consume.

This section describes how to extend a core concept within the debugger. Extensions which are built to be shared should conform to the guidelines presented in Native Debugger Objects in JavaScript Extensions - Design and Testing Considerations.

Registering an Extension

A script can register the fact that it provides an extension through an entry in the array returned from the initializeScript method.

```
function initializeScript()
{
    return [new host.namedModelParent(comProcessExtension, "Debugger.Models.Process")];
}
```

The presence of a host.namedModelParent object within the returned array indicates to the debugger that a given prototype object or ES6 class (comProcessExtension in this case) is going to be a parent data model to the

model which is registered under the name Debugger.Models.Process.

Debugger Object Extension Points

The following debugger extension points are integral to the debugger and available to be used by script providers such as JavaScript.

Debugger.Models.Sessions	The list of sessions (targets) that the debugger is attached to
Debugger.Models.Session	An individual session (target) that the debugger is attached to (live user mode, KD, etc...)
Debugger.Models.Processes	The list of processes within a session
Debugger.Models.Threads	The list of threads within a process
Debugger.Models.Thread	An individual thread within a process (regardless of whether user or kernel mode)
Debugger.Models.Stack	The stack of a thread
Debugger.Models.StackFrames	The collection of frames which make up a stack
Debugger.Models.StackFrame	An individual stack frame within a stack
Debugger.Models.LocalVariables	The local variables within a stack frame
Debugger.Models.Parameters	The parameters for a call within a stack frame
Debugger.Models.Module	An individual module within the address space of a process

Addtional Data Model Objects

In addition, there are some additional data model objects that are defined by the core data model.

DataModel.Models.Intrinsic	An intrinsic value (ordinals, floats, etc...)
DataModel.Models.String	A string
DataModel.Models.Array	A native array
DataModel.Models.Guid	A GUID
DataModel.Models.Error	An error object
DataModel.Models.Concepts.Iterable	Applied to every object which is iterable
DataModel.Models.Concepts.StringDisplayable	Applied to every object which has a display string conversion

Example COM Debugger Object Extension Overview

Let's consider an example. Imagine that you want to create a debugger extension to display information specific to COM, such as the global interface table (GIT).

In the past, there might be an existing debugger extension with a number of commands which provide a means to access things about COM. One command might display process centric information (the global interface table for instance). Another command might provide thread centric information such as what apartment code is executing within. You might need to know about and load a second debugger extension to explore other aspects of COM.

Instead of having a set of hard to discover commands, a JavaScript extension can modify the debugger's concept of what a process and a thread is, to add this information in a way that's natural, explorable, and composable with other debugger extensions.

User or Kernel Mode Debugger Object Extension

The debugger and the debugger objects have different behavior in user and kernel mode. When you create your debugger model objects you need to decide which environments you will be working in. Because we will be working with COM in user mode, we will create and test this com extension in user mode. In other situations, you may be able to create a debugger JavaScript that will work in both user and kernel mode debugging.

Creating a Sub-Namespace

Going back to our example, we can define a prototype or ES6 class, *comProcessExtension* which contains the set of things we want to add to a process object.

Important The intent with the sub-namespace is to create a logically structured and naturally explorable paradigm. For example, avoid dumping unrelated items into the same sub-namespace. Carefully review the information discussed in Native Debugger Objects in JavaScript Extensions - Design and Testing Considerations before creating a sub-namespace.

In this code snippet, we create add a sub-namespace called 'COM' on to the existing process debugger object.

```
var comProcessExtension =
{
    //
    // Add a sub-namespace called 'COM' on process.
    //
    get COM()
    {
        //
        // What is 'this' below...?  It's the debugger's process object.  Yes -- this means that there is a
cross-language
        // object hierarchy here.  A C++ object implemented in the debugger has a parent model (prototype)
which is
        // implemented in JavaScript.
        //
        return new comNamespace(this);
    }
}
```

Namespace Implementation

Next, create the object which implements the sub-namespace COM on a process.

Important There can be multiple processes (whether attached to such in user mode or under KD). This extension cannot assume that the present state of the debugger is the what the user intended. Someone can capture <someProcess>.COM in a variable and modify it, which can lead to presenting information from the wrong process context. The solution is to add code in the extension so that each instantiation will keep track of what process it is attached to. For this code sample, this information is passed via the 'this' pointer of the property.

```
this.__process = process;
```

```
class comNamespace
{
    constructor(process)
    {
        //
        // This is an entirely JavaScript object.  Each instantiation of a comNamespace will keep track
        // of what process it is attached to (passed via the ''this'' pointer of the property getter
        // we authored above.
        //
        this.__process = process;
    }

    get GlobalObjects()
    {
        return new globalObjects(this.__process);
    }
}
```

Implementation logic for the COM global interface table

To separate this out the implementation logic for the COM global interface table more clearly, we'll define one ES6 class, *gipTable* which abstracts away the COM GIP table and another, *globalObjects*, which is what will get returned from the GlobalObjects() getter defined in the Namespace Implementation code snip shown above. All of these details can be hidden inside the closure of initializeScript to avoid publishing any of these internal details out into the debugger namespace.

```
// gipTable:
//
// Internal class which abstracts away the GIP Table.  It iterates objects of the form
// {entry : GIPEntry, cookie : GIT cookie}
//
class gipTable
{
    constructor(gipProcess)
    {
        //
        // Windows 8 through certain builds of Windows 10, it's in CGIPTable::_palloc.  In certain builds
        // of Windows 10 and later, this has been moved to GIPEntry::_palloc.  We need to check which.
        //
        var gipAllocator = undefined;
        try
        {
            gipAllocator = host.getModuleSymbol("combase.dll", "CGIPTable::_palloc", "CPageAllocator",
gipProcess)._pgalloc;
        }
        catch(err)
        {
        }

        if (gipAllocator == undefined)
        {
            gipAllocator = host.getModuleSymbol("combase.dll", "GIPEntry::_palloc", "CPageAllocator",
gipProcess)._pgalloc;
        }

        this.__data = {
            process : gipProcess,
            allocator : gipAllocator,
            pageList : gipAllocator._pPageListStart,
            pageCount : gipAllocator._cPages,
            entriesPerPage : gipAllocator._cEntriesPerPage,
            bytesPerEntry : gipAllocator._cbPerEntry,
            PAGESHIFT : 16,
            PAGEMASK : 0x0000FFFF,
            SEQNOMASK : 0xFF00
```

```
        };
    }

    *[Symbol.iterator]()
    {
        for (var pageNum = 0; pageNum < this.__data.pageCount; ++pageNum)
        {
            var page = this.__data.pageList[pageNum];
            for (var entryNum = 0; entryNum < this.__data.entriesPerPage; ++entryNum)
            {
                var entryAddress = page.address.add(this.__data.bytesPerEntry * entryNum);
                var gipEntry = host.createPointerObject(entryAddress, "combase.dll", "GIPEntry *",
this.__data.process);
                if (gipEntry.cUsage != -1 && gipEntry.dwType != 0)
                {
                    yield {entry : gipEntry, cookie : (gipEntry.dwSeqNo | (pageNum << this.__data.PAGESHIFT)
| entryNum)};
                }
            }
        }
    }

    entryFromCookie(cookie)
    {
        var sequenceNo = (cookie & this.__data.SEQNOMASK);
        cookie = cookie & ~sequenceNo;
        var pageNum = (cookie >> this.__data.PAGESHIFT);
        if (pageNum < this.__data.pageCount)
        {
            var page = this.__data.pageList[pageNum];
            var entryNum = (cookie & this.__data.PAGEMASK);
            if (entryNum < this.__data.entriesPerPage)
            {
                var entryAddress = page.address.add(this.__data.bytesPerEntry * entryNum);
                var gipEntry = host.createPointerObject(entryAddress, "combase.dll", "GIPEntry *",
this.__data.process);
                if (gipEntry.cUsage != -1 && gipEntry.dwType != 0 && gipEntry.dwSeqNo == sequenceNo)
                {
                    return {entry : gipEntry, cookie : (gipEntry.dwSeqNo | (pageNum <<
this.__data.PAGESHIFT) | entryNum)};
                }
            }
        }

        //
        // If this exception flows back to C/C++, it will be a failed HRESULT (according to the type of
error -- here E_BOUNDS)
        // with the message being encapsulated by an error object.
        //
        throw new RangeError("Unable to find specified value");
    }
}




// globalObjects:
//
// The class which presents how we want the GIP table to look to the data model.  It iterates the actual
objects
// in the GIP table indexed by their cookie.
//
class globalObjects
{
    constructor(process)
    {
        this.__gipTable = new gipTable(process);
    }
```

```
    }

    *[Symbol.iterator]()
    {
        for (var gipCombo of this.__gipTable)
        {
            yield new host.indexedValue(gipCombo.entry.pUnk, [gipCombo.cookie]);
        }
    }

    getDimensionality()
    {
        return 1;
    }

    getValueAt(cookie)
    {
        return this.__gipTable.entryFromCookie(cookie).entry.pUnk;
    }
}
```

Lastly, use host.namedModelRegistration to register the new COM functionality.

```
function initializeScript()
{
    return [new host.namedModelParent(comProcessExtension, "Debugger.Models.Process"),
            new host.namedModelRegistration(comNamespace, "Debugger.Models.ComProcess")];
}
```

Save the code to GipTableAbstractor.js using an application such as notepad.

Here is the process information available in user mode before loading this extension.

```
0:000:x86> dx @$curprocess
@$curprocess                 : DataBinding.exe
    Name             : DataBinding.exe
    Id               : 0x1b9c
    Threads
    Modules
```

Load the JavaScript scripting provider and the extension.

```
0:000:x86> !load jsprovider.dll
0:000:x86> .scriptload C:\JSExtensions\GipTableAbstractor.js
JavaScript script successfully loaded from 'C:\JSExtensions\GipTableAbstractor.js'
```

Then use the dx command to display information about the process using the predefined @$curprocess.

```
0:000:x86> dx @$curprocess
@$curprocess                 : DataBinding.exe
    Name             : DataBinding.exe
    Id               : 0x1b9c
    Threads
    Modules
    COM              : [object Object]
```

```
0:000:x86> dx @$curprocess.COM
@$curprocess.COM                 : [object Object]
    GlobalObjects    : [object Object]
0:000:x86> dx @$curprocess.COM.GlobalObjects
@$curprocess.COM.GlobalObjects                 : [object Object]
    [0x100]          : 0x12f4fb0 [Type: IUnknown *]
    [0x201]          : 0x37cfc50 [Type: IUnknown *]
    [0x302]          : 0x37ea910 [Type: IUnknown *]
    [0x403]          : 0x37fcfe0 [Type: IUnknown *]
    [0x504]          : 0x12fe1d0 [Type: IUnknown *]
    [0x605]          : 0x59f04e8 [Type: IUnknown *]
    [0x706]          : 0x59f0eb8 [Type: IUnknown *]
    [0x807]          : 0x59f5550 [Type: IUnknown *]
    [0x908]          : 0x12fe340 [Type: IUnknown *]
    [0xa09]          : 0x5afcb58 [Type: IUnknown *]
```

This table is also programmatically accessible via GIT cookie.

```
0:000:x86> dx @$curprocess.COM.GlobalObjects[0xa09]
@$curprocess.COM.GlobalObjects[0xa09]                 : 0x5afcb58 [Type: IUnknown *]
    [+0x00c] __abi_reference_count [Type: __abi_FTMWeakRefData]
    [+0x014] __capture        [Type: Platform::Details::__abi_CapturePtr]
```

Extending Debugger Object Concepts with LINQ

In addition to being able to extend objects like process and thread, JavaScript can extend concepts associated with the data model as well. For example, it is possible to add a new LINQ method to every iterable. Consider an example extension, "DuplicateDataModel" which duplicates every entry in an iterable N times. The following code shows how this could be implemented.

```
function initializeScript()
{
    var newLinqMethod =
    {
        Duplicate : function *(n)
        {
            for (var val of this)
            {
                for (var i = 0; i < n; ++i)
                {
                    yield val;
                }
            };
        }
    };

    return [new host.namedModelParent(newLinqMethod, "DataModel.Models.Concepts.Iterable")];
}
```

Save the code to DuplicateDataModel.js using an application such as notepad.

Load the JavaScript scripting provider if necessary and then load the DuplicateDataModel.js extension.

```
0:000:x86> !load jsprovider.dll
0:000:x86> .scriptload C:\JSExtensions\DuplicateDataModel.js
JavaScript script successfully loaded from 'C:\JSExtensions\DuplicateDataModel.js'
```

Use the dx command to test the new Duplicate function.

```
0: kd> dx -r1 Debugger.Sessions.First().Processes.First().Threads.Duplicate(2),d
Debugger.Sessions.First().Processes.First().Threads.Duplicate(2),d                : [object Generator]
    [0]              : nt!DbgBreakPointWithStatus (fffff800`9696ca60)
    [1]              : nt!DbgBreakPointWithStatus (fffff800`9696ca60)
    [2]              : intelppm!MWaitIdle+0x18 (fffff805`0e351348)
    [3]              : intelppm!MWaitIdle+0x18 (fffff805`0e351348)
…
```

Related topics

Native Debugger Objects in JavaScript Extensions - Debugger Object Details

Native Debugger Objects in JavaScript Extensions - Design and Testing Considerations

JavaScript Debugger Scripting

JavaScript Debugger Example Scripts

Native Debugger Objects in JavaScript Extensions - Debugger Object Details

9/12/2019 • 19 minutes to read • Edit Online

This topic describes additional details about using the native debugger objects in JavaScript extensions.

Native debugger objects represent various constructs and behaviors of the debugger environment. The objects can be passed into (or acquired in) JavaScript extensions to manipulate the state of the debugger.

For information about Debugger object JavaScript extensions, see Native Debugger Objects in JavaScript Extensions.

For general information about working with JavaScript, see JavaScript Debugger Scripting.

Debugger Objects in JavaScript Extensions

Passing Native Objects

Debugger objects can be passed into or acquired in JavaScript extensions in a variety of ways.

- They can be passed to JavaScript functions or methods
- They can be the instance object for a JavaScript prototype (as a visualizer, for instance)
- They can be returned from host methods designed to create native debugger objects
- They can be returned from host methods designed to create debugger native objects

Debugger objects that are passed to a JavaScript extension have a set of functionality that is described in this section.

- Property Access
- Projected Names
- Special Types Pertaining to Native Debugger Objects
- Additional Attributes

Property Access

While there are some properties on objects which are placed there by the JavaScript provider itself, the majority of properties on a native object which enters JavaScript are provided by the data model. This means that for a property access --- object.propertyName or object[propertyName], the following will occur.

- If *propertyName* is the name of a property projected onto the object by the JavaScript provider itself, it will resolve to this first; otherwise
- If *propertyName* is the name of a key projected onto the object by the data model (another Visualizer), it will resolve to this name second; otherwise
- If *propertyName* is the name of a field of the native object, it will resolve to this name third; otherwise
- If object is a pointer, the pointer will be dereferenced, and the cycle above will continue (a projected property of the dereferenced object followed by a key followed by a native field)

The normal means of property access in JavaScript -- object.propertyName and object[propertyName] -- will access the underlying native fields of an object, much as the 'dx' command would within the debugger.

Projected Names

The following properties (and methods) are projected onto native objects which enter JavaScript.

METHOD	SIGNATURE	DESCRIPTION
hostContext	Property	Returns an object which represents the context the object is within (the address space, debug target, etc...)
targetLocation	Property	Returns an object which is an abstraction of where the object is within an address space (virtual address, register, sub-register, etc...)
targetSize	Property	Returns the size of the object (effectively: sizeof(<TYPE OF OBJECT>)
addParentModel	.addParentModel(object)	Adds a new parent model (akin to a JavaScript prototype but on the data mdoel side) to the object
removeParentModel	.removeParentModel(object)	Removes a given parent model from the object
runtimeTypedObject	Property	Performs analysis on the object and tries to convert it to the runtime (most derived) type

If the object is a pointer, the following properties (and methods) are projected onto the pointer which enters JavaScript:

PROPERTY NAME	SIGNATURE	DESCRIPTION
add	.add(value)	Performs pointer math addition between the pointer and the specified value
address	Property	Returns the address of the pointer as a 64-bit ordinal object (a library type)
dereference	.dereference()	Dereferences the pointer and returns the underlying object
isNull	Property	Returns whether or not the pointer value is nullptr (0)

Special Types Pertaining to Native Debugger Objects

Location Objects

The location object which is returned from the targetLocation property of a native object contains the following properties (and methods).

PROPERTY NAME	SIGNATURE	DESCRIPTION
add	.add(value)	Adds an absolute byte offset to the location.

PROPERTY NAME	SIGNATURE	DESCRIPTION
subtract	.subtract(value)	Subtracts an absolute byte offset from the location.

Additional Attributes

Iterability

Any object which is understood as iterable by the data model (it is a native array or it has a visualizer (NatVis or otherwise) which makes it iterable) will have an iterator function (indexed via the ES6 standard Symbol.iterator) placed upon it. This means that you can iterate a native object in JavaScript as follows.

```
function iterateNative(nativeObject)
{
    for (var val of nativeObject)
    {
        //
        // val will contain each element iterated from the native object.  This would be each element of an
array,
        // each element of an STL structure which is made iterable through NatVis, each element of a data
structure
        // which has a JavaScript iterator accessible via [Symbol.iterator], or each element of something
        // which is made iterable via support of IIterableConcept in C/C++.
        //
    }
}
```

Indexability

Objects which are understood as indexable in one dimension via ordinals (e.g.: native arrays) will be indexable in JavaScript via the standard property access operator -- object[index]. If an object is indexable by name or is indexable in more than one dimension, the getValueAt and setValueAt methods will be projected onto the object so that JavaScript code can utilize the indexer.

```
function indexNative(nativeArray)
{
    var first = nativeArray[0];
}
```

String Conversion

Any native object which has a display string conversion via support of IStringDisplayableConcept or a NatVis DisplayString element will have that string conversion accessible via the standard JavaScript toString method.

```
function stringifyNative(nativeObject)
{
    var myString = nativeObject.toString();
}
```

Creating Native Debugger Objects

As mentioned, a JavaScript script can get access to native objects by having them passed into JavaScript in one of several ways or it can create them through calls to the host library. Use the following functions to create native debugger objects.

METHOD	SIGNATURE	DESCRIPTION
host.getModuleSymbol	getModuleSymbol(moduleName, symbolName, [contextInheritor]) getModuleSymbol(moduleName, symbolName, [typeName], [contextInheritor])	Returns an object for a global symbol within a particular module. The module name and symbol name are strings. If the optional *contextInheritor* argument is supplied, the module and symbol will be looked up within the same context (address space, debug target) as the passed object. If the argument is not supplied, the module and symbol will be looked up in the debugger's current context. A JavaScript extension which is not a one-off test script should always supply an explicit context. If the optional *typeName* argument is supplied, the symbol will be assumed to be of the passed type and the type indicated in symbol(s) will be ignored. Note that any caller which expects to operate on public symbols for a module should always supply an explicit type name.
host.createPointerObject	createPointerObject(address, moduleName, typeName, [contextInheritor])	Creates a pointer object at the specified address or location. The module name and type name are strings. If the optional *contextInheritor* argument is supplied, the module and symbol will be looked up within the same context (address space, debug target) as the passed object. If the argument is not supplied, the module and symbol will be looked up in the debugger's current context. A JavaScript extension which is not a one-off test script should always supply an explicit context.

METHOD	SIGNATURE	DESCRIPTION
host.createTypedObject	createTypedObject(location, moduleName, typeName, [contextInheritor])	Creates a object which represents a native typed object within the address space of a debug target at the specified location. The module name and type name are strings. If the optional *contextInheritor* argument is supplied, the module and symbol will be looked up within the same context (address space, debug target) as the passed object. If the argument is not supplied, the module and symbol will be looked up in the debugger's current context. A JavaScript extension which is not a one-off test script should always supply an explicit context.

Host APIs for JavaScript Extensions

The JavaScript provider inserts an object called host into the global namespace of every script which it loads. This object provides access to critical functionality for the script as well as access to the namespace of the debugger. It is set up in two phases.

- **Phase 1**: Before any script executes, the host object only contains the minimal set of functionality necessary for a script to initialize itself and register its extensibility points (both as producer and consumer). The root and initialization code is not intended to manipulate the state of a debug target or perform complex operations and, as such, the host is not fully populated until after the initializeScript method returns.
- **Phase 2**: After initializeScript returns, the host object is populated with everything necessary to manipulate the state of debug targets.

Host Object Level

A few key pieces of functionality are directly under the host object. The remainder are sub-namespaced. Namespaces include the following.

NAMESPACE	DESCRIPTION
diagnostics	Functionality to assist in the diagnosis and debugging of script code
memory	Functionality to enable memory reading and writing within a debug target

Root Level

Directly within the host object, the following properties, methods, and constructors can be found.

NAME	SIGNATURE	PHASE PRESENT	DESCRIPTION

NAME	SIGNATURE	PHASE PRESENT	DESCRIPTION
createPointerObject	createPointerObject(address, moduleName, typeName, [contextInheritor])	2	Creates a pointer object at the specified address or location. The module name and type name are strings. The optional **contextInheritor** argument works as with getModuleSymbol.
createTypedObject	createTypedObject(location, moduleName, typeName, [contextInheritor])	2	Creates a object which represents a native typed object within the address space of a debug target at the specified location. The module name and type name are strings. The optional contextInheritor argument works as with getModuleSymbol.
currentProcess	Property	2	Returns the object representing the current process of the debugger
currentSession	Property	2	Returns the object representing the current session of the debugger (which target, dump, etc...) is being debugged
currentThread	Property	2	Returns the object representing the current thread of the debugger
evaluateExpression	evaluateExpression(expression, [contextInheritor])	2	This calls into the debug host to evaluate an expression using the language of the debug target only. If the optional *contextInheritor* argument is supplied, the expression will be evaluated in the context (e.g.: address space and debug target) of the argument; otherwise, it will be evaluated in the current context of the debugger

NAME	SIGNATURE	PHASE PRESENT	DESCRIPTION
evaluateExpressionInContext	evaluateExpressionInContext(context, expression)	2	This calls into the debug host to evaluate an expression using the language of the debug target only. The context argument indicates the implicit this pointer to utilize for the evaluation. The expression will be evaluated in the context (e.g.: address space and debug target) indicated by the *context* argument.
getModuleSymbol	getModuleSymbol(moduleName, symbolName, [contextInheritor])	2	Returns an object for a global symbol within a particular module. The module name and symbol name are strings. If the optional *contextInheritor* argument is supplied, the module and symbol will be looked up within the same context (address space, debug target) as the passed object. If the argument is not supplied, the module and symbol will be looked up in the debugger's current context. A JavaScript extension which is not a one-off script should always supply an explicit context
getNamedModel	getNamedModel(modelName)	2	Returns the data model which was registered against a given name. Note that it is perfectly legal to call this against a name which is not yet registered. Doing so will create a stub for that name and manipulations of the stub will be made to the actual object upon registration
indexedValue	new indexedValue(value, indicies)	2	A constructor for an object which can be returned from a JavaScript iterator in order to assign a default set of indicies to the iterated value. The set of indicies must be expressed as a JavaScript array.

NAME	SIGNATURE	PHASE PRESENT	DESCRIPTION
Int64	new Int64(value, [highValue])	1	This constructs a library Int64 type. The single argument version will take any value which can pack into an Int64 (without conversion) and place it into such. If an optional second argument is supplied, a conversion of the first argument is packed into the lower 32-bits and a conversion of the second argument is packed into the upper 32 bits.
namedModelParent	new namedModelParent(object, name)	1	A constructor for an object intended to be placed in the array returned from **initializeScript**, this represents using a JavaScript prototype or ES6 class as a data model parent extension of a data model with the given name
namedModelRegistration	new namedModelRegistration(object, name)	1	A constructor for an object intended to be placed in the array returned from **initializeScript**, this represents the registration of a JavaScript prototype or ES6 class as a data model via a known name so that other extensions can find and extend
namespace	Property	2	Gives direct access to the root namespace of the debugger. One could, for example, access the process list of the first debug target via host.namespace.Debugger.Sessions.First().Processes using this property

NAME	SIGNATURE	PHASE PRESENT	DESCRIPTION
registerNamedModel	registerNamedModel(object, modelName)	2	This registers a JavaScript prototype or ES6 class as a data model under the given name. Such a registration allows the prototype or class to be located and extended by other scripts or other debugger extensions. Note that a script should prefer to return a **namedModelRegistration** object from its **initializeScript** method rather than doing this imperatively. Any script which makes changes imperatively is required to have an **initializeScript** method in order to clean up.
registerExtensionForTypeSignature	registerExtensionForTypeSignature(object, typeSignature)	2	This registers a JavaScript prototype or ES6 class as an extension data model for a native type as given by the supplied type signature. Note that a script should prefer to return a **typeSignatureExtension** object from its **initializeScript** method rather than doing this imperatively. Any script which makes changes imperatively is required to have an **initializeScript** method in order to clean up.
registerPrototypeForTypeSignature	registerPrototypeForTypeSignature(object, typeSignature)	2	This registers a JavaScript prototype or ES6 class as the canonical data model (e.g.: visualizer) for a native type as given by the supplied type signature. Note that a script should prefer to return a **typeSignatureExtension** object from its **initializeScript** method rather than doing this imperatively. Any script which makes changes imperatively is required to have an **uninitializeScript**method in order to clean up.

NAME	SIGNATURE	PHASE PRESENT	DESCRIPTION
parseInt64	parseInt64(string, [radix])	1	This method acts similarly to the standard JavaScript parseInt method except that it returns a library Int64 type instead. If a radix is supplied, the parse will occur in either base 2, 8, 10, or 16 as indicated.
typeSignatureExtension	new typeSignatureExtension(object, typeSignature, [moduleName], [minVersion], [maxVersion])	1	A constructor for an object intended to be placed in the array returned from **initializeScript**, this represents an extension of a native type described via a type signature by a JavaScript prototype or ES6 class. Such a registration "adds fields" to the debugger's visualization of any type which matches the signature rather than taking it over entirely. An optional module name and version can restrict the registration. Versions are specified as "1.2.3.4" style strings.
typeSignatureRegistration	new typeSignatureRegistratio n(object, typeSignature, [moduleName], [minVersion], [maxVersion])	1	A constructor for an object intended to be placed in the array returned from **initializeScript**, this represents a canonical registration of a JavaScript prototype or ES6 class against a native type signature. Such a registration "takes over" the debugger's visualization of any type which matches the signature rather than merely than extending it. An optional module name and version can restrict the registration. Versions are specified as "1.2.3.4" style strings.
unregisterNamedModel	unregisterNamedModel(modelName)	2	This unregisters a data model from lookup by the given name undoing any operation performed by **registerNamedModel**

NAME	SIGNATURE	PHASE PRESENT	DESCRIPTION
unregisterExtensionForTypeSignature	unregisterExtensionForTypeSignature(object, typeSignature, [moduleName], [minVersion], [maxVersion])	2	This unregisters a JavaScript prototype or ES6 class from being an extension data model for a native type as given by the supplied type signature. It is the logical undo of registerExtensionForTypeSignature. Note that a script should prefer to return a **typeSignatureExtension** object from its **initializeScript** method rather than doing this imperatively. Any script which makes changes imperatively is required to have an **initializeScript** method in order to clean up. An optional module name and version can restrict the registration. Versions are specified as "1.2.3.4" style strings.
unregisterPrototypeForTypeSignature	unregisterPrototypeForTypeSignature(object, typeSignature, [moduleName], [minVersion], [maxVersion])	2	This unregisters a JavaScript prototype or ES6 class from being the canonical data model (e.g.: visualizer) for a native type as given by the supplied type signature. It is the logical undo of registerPrototypeForTypeSignature. Note that a script should prefer to return a **typeSignatureRegistration** object from its **initializeScript** method rather than doing this imperatively. Any script which makes changes imperatively is required to have an **uninitializeScript** method in order to clean up. An optional module name and version can restrict the registration. Versions are specified as "1.2.3.4" style strings.

Diagnostics Functionality

The diagnostics sub-namespace of the host object contains the following.

NAME	SIGNATURE	PHASE PRESENT	DESCRIPTION

NAME	SIGNATURE	PHASE PRESENT	DESCRIPTION
debugLog	debugLog(object...)	1	This provides printf style debugging to a script extension. At present, output from debugLog is routed to the output console of the debugger. At a later point in time, there are plans to provide flexibility on routing this output. NOTE: This should not be used as a means of printing user output to console. It may not be routed there in the future.

Memory Functionality

The memory sub-namespace of the host object contains the following.

NAME	SIGNATURE	PHASE PRESENT	DESCRIPTION
readMemoryValues	readMemoryValues(location, numElements, [elementSize], [isSigned], [contextInheritor])	2	This reads a raw array of values from the address space of the debug target and places a typed array on top of the view of this memory. The supplied location can be an address (a 64-bit value), a location object, or a native pointer. The size of the array is indicated by the *numElements* argument. The size (and type) of each element of the array is given by the optional *elementSize* and *isSigned* arguments. If no such arguments are supplied, the default is byte (unsigned / 1 byte). If the optional *contextInheritor* argument is supplied, memory will be read in the context (e.g.: address space and debug target) indicated by the argument; otherwise, it will be read from the debugger's current context. Note that using this method on 8, 16, and 32-bit values results in a fast typed view being placed over the read memory. Using this method on 64-bit values results in an array of 64-bit library types being constructed which is significantly more expensive!

NAME	SIGNATURE	PHASE PRESENT	DESCRIPTION
readString	readString(location, [contextInheritor]) readString(location, [length], [contextInheritor])	2	This reads a narrow (current code page) string from the address space of a debug target, converts it to UTF-16, and returns the result as a JavaScript string. It may throw an exception if the memory could not be read. The supplied location can be an address (a 64-bit value), a location object, or a native char. *If the optional contextInheritor argument is supplied, memory will be read in the context (e.g.: address space and debug target) indicated by the argument; otherwise, it will be read from the debugger's current context. If the optional length argument is supplied, the read string will be of the specified length.*
readWideString	readWideString(location, [contextInheritor]) readWideString(location, [length], [contextInheritor])	2	This reads a wide(UTF-16) string from the address space of a debug target and returns the result as a JavaScript string. It may throw an exception if the memory could not be read. The supplied location can be an address (a 64-bit value), a location object, or a native wchar_t. If the optional *contextInheritor* argument is supplied, memory will be read in the context (e.g.: address space and debug target) indicated by the argument; otherwise, it will be read from the debugger's current context. If the optional *length* argument is supplied, the read string will be of the specified length.

Data Model Concepts in JavaScript

Data Model Mapping

The following data model concepts map to JavaScript.

CONCEPT	NATIVE INTERFACE	JAVASCRIPT EQUIVALENT
String Conversion	IStringDisplayableConcept	standard: toString(...){...}

CONCEPT	NATIVE INTERFACE	JAVASCRIPT EQUIVALENT
Iterability	IIterableConcept	standard: [Symbol.iterator](){...}
Indexability	IIndexableConcept	protocol: getDimensionality(...) / getValueAt(...) / setValueAt(...)
Runtime Type Conversion	IPreferredRuntimeTypeConcept	protocol: getPreferredRuntimeTypedObject(...)

String Conversion

The string conversion concept (IStringDisplayableConcept) directly translates to the standard JavaScript **toString** method. As all JavaScript objects have a string conversion (provided by Object.prototype if not provided elsewhere), every JavaScript object returned to the data model can be converted to a display string. Overriding the string conversion simply requires implementing your own toString.

```
class myObject
{
    //
    // This method will be called whenever any native code calls
IStringDisplayableConcept::ToDisplayString(...)
    //
    toString()
    {
        return "This is my own string conversion!";
    }
}
```

Iterability

The data model's concept of whether an object is iterable or not maps directly to the ES6 protocol of whether an object is iterable. Any object which has a [Symbol.iterator] method is considered iterable. Implementation of such will make the object iterable.

An object which is only iterable can have an implementation such as follows.

```
class myObject
{
    //
    // This method will be called whenever any native code calls IIterableConcept::GetIterator
    //
    *[Symbol.iterator]()
    {
        yield "First Value";
        yield "Second Value";
        yield "Third Value";
    }
}
```

Special consideration must be given for objects which are both iterable and indexable as the objects returned from the iterator must include the index as well as the value via a special return type.

Iterable and Indexable

An object which is iterable and indexable requires a special return value from the iterator. Instead of yielding the values, the iterator yields instances of indexedValue. The indicies are passed as an array in the second argument to the indexedValue constructor. They can be multi-dimensional but must match the dimensionality returned in the indexer protocol.

This code shows an example implementaion.

```
class myObject
{
    //
    // This method will be called whenever any native code calls IIterableConcept::GetIterator
    //
    *[Symbol.iterator]()
    {
        //
        // Consider this a map which mapped 42->"First Value", 99->"Second Value", and 107->"Third Value"
        //
        yield new host.indexedValue("First Value", [42]);
        yield new host.indexedValue("Second Value", [99]);
        yield new host.indexedValue("Third Value", [107]);
    }
}
```

Indexability

Unlike JavaScript, the data model makes a very explicit differentiation between property access and indexing. Any JavaScript object which wishes to present itself as indexable in the data model must implement a protocol consisting of a getDimensionality method which returns the dimensionality of the indexer and an optional pair of getValueAt and setValueAt methods which perform reads and writes of the object at supplied indicies. It is acceptable to omit either the getValueAt or setValueAt methods if the object is read-only or write-only

```
class myObject
{
    //
    // This method will be called whenever any native code calls IIndexableConcept::GetDimensionality or
IIterableConcept::GetDefaultIndexDimensionality
    //
    getDimensionality()
    {
        //
        // Pretend we are a two dimensional array.
        //
        return 2;
    }

    //
    // This method will be called whenever any native code calls IIndexableConcept::GetAt
    //
    getValueAt(row, column)
    {
        return this.__values[row * this.__columnCount + column];
    }

    //
    // This method will be called whenever any native code calls IIndexableConcept::SetAt
    //
    setValueAt(value, row, column)
    {
        this.__values[row * this.__columnCount + column] = value;
    }
}
```

Runtime Type Conversion

This is only relevant for JavaScript prototypes/classes which are registered against type system (native) types. The debugger is often capable of performing analysis (e.g. Run-Time Type Information (RTTI) / v-table analysis) to determine the true runtime type of an object from a static type expressed in code. A data model registered against

a native type can override this behavior via an implementation of the IPreferredRuntimeTypeConcept. Likewise, a JavaScript class or prototype registered against a native object can provide its own implementation via implementation of a protocol consisting of the getPreferredRuntimeTypedObject method.

Note that while this method can technically return anything, it is considered bad form for it to return something which isn't really the runtime type or a derived type. Such can result in significant confusion for users of the debugger. Overriding this method can, however, be valuable for things such as C-style header+object styles of implementation, etc...

```
class myNativeModel
{
    //
    // This method will be called whenever the data model calls
IPreferredRuntimeTypeConcept::CastToPreferredRuntimeType
    //
    getPreferredRuntimeTypedObject()
    {
        var loc = this.targetLocation;

        //
        // Perform analysis...
        //
        var runtimeLoc = loc.Add(runtimeObjectOffset);

        return host.createTypedObject(runtimeLoc, runtimeModule, runtimeTypeName);
    }
}
```

Related topics

Native Debugger Objects in JavaScript Extensions

Native Debugger Objects in JavaScript Extensions - Design and Testing Considerations

JavaScript Debugger Scripting

JavaScript Debugger Example Scripts

Native Debugger Objects in JavaScript Extensions - Design and Testing Considerations

9/12/2019 • 4 minutes to read • Edit Online

This topic describes design and testing considerations for using the native debugger objects in JavaScript extensions.

Native debugger objects represent various constructs and behaviors of the debugger environment. The objects can be passed into (or acquired in) JavaScript extensions to manipulate the state of the debugger.

For information about Debugger object JavaScript extensions, see Native Debugger Objects in JavaScript Extensions.

For general information about working with JavaScript, see JavaScript Debugger Scripting.

Debugger Data Model Design Considerations

Design Principles

Consider the following principles to make your debugger extensions present information that is discoverable, queryable, and scriptable.

- Information is close to where it is needed. For example, information on a registry key should be displayed as part of a local variable that contains a registry key handle.
- Information is structured. For example, information about a registry key is presented in separate fields such as key type, key ACL, key name, and value. This means that the individual fields can be accessed without parsing text.
- Information is consistent. Information about registry key handles is presented in as similar a way as possible to information about file handles.

Avoid these approaches that do not support these principles.

- Do not structure your items into a single flat "Kitchen sink". An organized hierarchy allows users to browse for the information they are looking for without prior knowledge of what they are looking for and supports discoverability.
- Do not convert a classic dbgeng extension by simply moving it to the model while still outputting screens of raw text. This is not composable with other extensions and cannot be queried with LINQ expressions. Instead break the data into separate, queryable fields.

Naming Guidelines

- Capitalization of fields should be PascalCase. An exception could be considered for names that are widely known in another casing, such as jQuery.
- Avoid using special characters that would not normally be used in a C++ identifier. For example, avoid using names such as "Total Length" (that contains a space), or "[size]" (that contains square brackets). This convention allows for easier consumption from scripting languages where these characters are not allowed as part of identifiers, and also allows easier consumption from the command window.

Organization and Hierarchy Guidelines

- Do not extend the top level of the debugger namespace. Instead, you should extend an existing node in the debugger so that the information is displayed where it is most relevant.

- Do not duplicate concepts. If you are creating a data model extension that lists additional information about a concept that already exists in the debugger, extend the existing information rather than trying to replace it with new information. In other words, an extension that displays details about a module should extend the existing *Module* object rather than creating a new list of modules.
- Free floating utility commands must be part of the *Debugger.Utility* namespace. They should also be sub-namespaced appropriately (e.g. *Debugger.Utility.Collections.FromListEntry*)

Backwards Compatibility and Breaking Changes

A script that is published should not break compatibility with other scripts that depend on it. For example, if a function is published to the model, it should remain in the same location and with the same parameters, whenever possible.

No Use of Outside Resources

- Extensions must not spawn external processes. External processes can interfere with the behavior of the debugger, and will misbehave in various remote debugger scenarios (e.g. dbgsrv remotes, ntsd remotes, and "ntsd -d remotes")
- Extensions must not display any user interface. Displaying user interface elements will behave incorrectly on remote debugging scenarios, and can break console debugging scenarios.
- Extensions must not manipulate the debugger engine or debugger UI through undocumented methods. This causes compatibility problems and will behave incorrectly on debugger clients with different UI.
- Extensions must access target information only through the documented debugger APIs. Trying to access information about a target through win32 APIs will fail for many remote scenarios, and even some local debugging scenarios across security boundaries.

No Use of Dbgeng Specific Features

Scripts that are intended to be used as extensions must not rely on dbgeng-specific features whenever possible (such as executing "classic" debugger extensions). Scripts should be usable on top of any debugger that hosts the data model.

Testing Debugger Extensions

Extensions are expected to work in a wide range of scenarios. While some extensions may be specific to a scenario (such as a kernel debugging scenario), most extensions should be expected to work in all scenarios, or have metadata indicating the supported scenarios.

Kernel Mode

- Live kernel debugging
- Kernel dump debugging

User Mode

- Live user mode debugging
- User mode dump debugging

In addition, consider these debugger usage scenarios

- Multi-process debugging
- Multi-session debugging (e.g. dump + live user within a single session)

Remote Debugger Usage

Test for proper operation with the remote debugger usage scenarios.

- dbgsrv remotes
- ntsd remotes
- ntsd -d remotes

For more information, see Debugging Using CDB and NTSD and **Activating a Process Server**.

Regression testing

Investigate the use of test automation that can verify the functionality of your extensions, as new versions of the debugger are released.

Related topics

Native Debugger Objects in JavaScript Extensions

Native Debugger Objects in JavaScript Extensions - Debugger Object Details.

JavaScript Debugger Scripting

JavaScript Debugger Example Scripts

Native Debugger Objects in NatVis

6/25/2019 • 2 minutes to read • Edit Online

Overview

Native debugger objects represent various constructs and behaviors of the debugger environment. Example debugger objects include the following.

- Session
- Threads / Thread
- Processes / Process
- Stack Frames / Stack Frame
- Local Variables
- Modules / Module
- Utility
- State
- Settings

You can use the dx command and LINQ to interact with the debugger objects. For more informaton, see dx (Display Debugger Object Model Expression) and Using LINQ With the debugger objects.

You can also work with debugger objects using JavaScript. For more information about that see, Native Debugger Objects in JavaScript Extensions.

This topic describes how you can create custom NatVis visualizers to display debugger objects.

NatVis Development Resources

Refer to these resources for general information about working with NatVis.

Create custom views of native objects

Writing debugger type visualizers for C++ using .natvis files

.nvload

.nvlist

.nvunload

.nvunloadall

Custom NatVis object example

Create a simple C++ application that has an instance of the class **CDog**.

```
class CDog
{
public:
   CDog(){m_age = 8; m_weight = 30;}
   long m_age;
   long m_weight;
};

int main()
{
   CDog MyDog;
   printf_s("%d, %d\n", MyDog.m_age, MyDog.m_weight);
   return 0;
}
```

Create a file named Dog.natvis that contains this XML:

```
<?xml version="1.0" encoding="utf-8"?>
<AutoVisualizer xmlns="https://schemas.microsoft.com/vstudio/debugger/natvis/2010">
   <Type Name="CDog">
      <DisplayString>{{Age = {m_age} years. Weight = {m_weight} pounds.}}</DisplayString>
   </Type>
</AutoVisualizer>
```

Copy Dog.natvis to the Visualizers folder in your installation directory for Debugging Tools for Windows. For example:

C:\Program Files\Debugging Tools for Windows (x64)\Visualizers

Run your program, and break in at the main function. Take a step so that the variable `MyDog` gets initialized. Display `MyDog` using **??** and again using **dx**.

```
0:000> ??MyDog
class CDog
   +0x000 m_age        : 0n8
   +0x004 m_weight     : 0n30
0:000> *
0:000> dx -r1 MyDog
.....
MyDog     : {Age = 8 years. Weight = 30 pounds.} [Type: CDog]
```

See also

dx (Display Debugger Object Model Expression)

Using LINQ With the debugger objects

Native Debugger Objects in JavaScript Extensions

Using LINQ With the debugger objects

10/23/2019 • 19 minutes to read • Edit Online

LINQ syntax can be used with the debugger objects to search and manipulate data. Using the LINQ syntax with dx command allows for a more consistent experience compared to using debugger commands. The output and options are consistent no matter which debugger object that you are looking at. LINQ queries allow you to ask questions such as "What are the top 5 processes that are running the most threads?".

Debugger objects are projected into a namespace rooted at "Debugger". Processes, modules, threads, stacks, stack frames, and local variables are all available to be used in a LINQ query.

LINQ is conceptually similar to the Structured Query Language (SQL) that is used to query databases. You can use a number of LINQ methods to search, filter and parse debug data. The LINQ C# method syntax is used. For more information on LINQ and the LINQ C# syntax, see Getting Started with LINQ in C#

LINQ that is used in the debugger support uses the “method syntax” of LINQ and not the “query syntax”. You can find more details about the differences in LINQ (Language-Integrated Query).

LINQ commands such as the following can be used with the debugger objects. All, .Any, .Count, .First, .Flatten, .GroupBy, .Last, .OrderBy, .OrderByDescending, .Select, and .Where. These methods follow (as closely as possible) the C# LINQ method form.

Native debugger objects

Native debugger objects represent various constructs and behaviors of the debugger environment. Example debugger objects include the following.

- Session
- Threads / Thread
- Processes / Process
- Stack Frames / Stack Frame
- Local Variables
- Modules / Module
- Utility
- State
- Settings

You can also work with the debugger objects with NatVis. For more information see Native Debugger Objects in NatVis. For information about using debugger objects with JavaScript, see Native Debugger Objects in JavaScript Extensions. For information on working with C++ and the driver objects, see Debugger Data Model C++ Overview.

Dx command

The examples shown here use the dx command, for more information about working with the dx command, see dx (Display Debugger Object Model Expression).

Developing a LINQ Query

One way to develop a LINQ debugger object query is to use the DML links that are displayed to explore the data model to first locate the debugger object that will be used in the query.

For this example, we would like to display a list of processes in a kernel debug session and the number of threads for each of those processes.

To start our exploration we can use the dx command to display the top level debugger object.

```
0: kd> dx Debugger
Debugger
    Sessions
    Settings
    State
    Utility
```

After clicking on the top level topics, we determine that Sessions looks most interesting, so we click on the DML link to reveal that it contains *Processes*.

```
0: kd> dx -r1 Debugger.Sessions[0]
Debugger.Sessions[0]                 : Remote KD: KdSrv:Server=@{<Local>},Trans=@{NET:Port=50005,Key=MyKey}
    Processes
    Id               : 0
    Attributes
```

Then we click further down to look at specific process and we see that the *Threads* associated with that process are available. When we click on *Threads* for one of the processes, we see the all of the threads associated with that process are available.

```
0: kd> dx -r1 Debugger.Sessions[0].Processes[1428].Threads
Debugger.Sessions[0].Processes[1428].Threads
    [0x598]          : <Unable to get stack trace> [Switch To]
    [0x1220]         : <Unable to get stack trace> [Switch To]
    [0x6f8]          : nt!KiSwapContext+0x76 (fffff806`4466a186)  [Switch To]
    [0x128c]         : <Unable to get stack trace> [Switch To]
    [0x27e4]         : nt!KiSwapContext+0x76 (fffff806`4466a186)  [Switch To]
```

We now know that the data that we need to display the number of threads associated with a process is available in the debugger object model.

To make the LINQ query a little shorter we can use the System Defined Variables described later in this topic to display the processes associated with the current session.

```
0: kd> dx @$cursession.Processes
@$cursession.Processes
    [0x0]            : Idle [Switch To]
    [0x4]            : System [Switch To]
    [0x90]           : Registry [Switch To]
...
```

Next add a select statement. To start with, we can specify the Name field.

```
0: kd> dx @$cursession.Processes.Select(p => p.Name)
@$cursession.Processes.Select(p => p.Name)
    [0x0]            : Idle
    [0x4]            : System
    [0x90]           : Registry
...
```

For our scenario, we also need the number of threads. Because there are two fields, create an anonymous type using *new*, similar to C#'s anonymous type syntax described below in User Defined Variables.

```
dx @$cursession.Processes.Select(p => new {Name = p.Name, Threads = p.Threads})
```

With that command, 'dx' doesn't actually print out the name anymore, so add -r2 (recurse two levels) to display Name and Threads.

```
dx -r2 @$cursession.Processes.Select(p => new {Name = p.Name, Threads = p.Threads})
@$cursession.Processes.Select(p => new {Name = p.Name, Threads = p.Threads})
    [0x0]
        Name             : Idle
        Threads
    [0x4]
        Name             : System
        Threads
    [0x90]
        Name             : Registry
        Threads
```

At this point we are displaying the name of the process and a list of threads. To display the ThreadCount, use the *.Count()* method.

```
0: kd> dx -r2 @$cursession.Processes.Select(p => new {Name = p.Name, ThreadCount = p.Threads.Count()})
@$cursession.Processes.Select(p => new {Name = p.Name, ThreadCount = p.Threads.Count()})
    [0x0]
        Name             : Idle
        ThreadCount      : 0x4
    [0x4]
        Name             : System
        ThreadCount      : 0xe7
    [0x90]
        Name             : Registry
        ThreadCount      : 0x4
...
```

To see which processes have a large number of threads, order the list by thread count using *OrderByDescending*.

```
0: kd> dx -r2 @$cursession.Processes.Select(p => new {Name = p.Name, ThreadCount =
p.Threads.Count()}).OrderByDescending(p => p.ThreadCount)
@$cursession.Processes.Select(p => new {Name = p.Name, ThreadCount = p.Threads.Count()}).OrderByDescending(p
=> p.ThreadCount)
    [0x4]
        Name             : System
        ThreadCount      : 0xe7
    [0xa38]
        Name             : svchost.exe
        ThreadCount      : 0x45
    [0x884]
        Name             : MemCompression
        ThreadCount      : 0x3e
```

To render in a formatted grid change the '-r2' to be '-g'. The level of recursion does not need to be specified, because the grid option displays the columns appropriately. Lastly, add the ',d' format specifier to output decimal values.

```
0: kd> dx -g @$cursession.Processes.Select(p => new {Name = p.Name, ThreadCount =
p.Threads.Count()}).OrderByDescending(p => p.ThreadCount),d
================================================================================================
=            = Name                                                            = ThreadCount =
================================================================================================
= [4]        - System                                                          - 231         =
= [2616]     - svchost.exe                                                     - 69          =
= [2180]     - MemCompression                                                  - 62          =
= [968]      - explorer.exe                                                    - 61          =
```

Debugger Objects Examples

This example shows the top 5 processes running the most threads:

```
0: kd> dx -r2 Debugger.Sessions.First().Processes.Select(p => new { Name = p.Name, ThreadCount =
p.Threads.Count() }).OrderByDescending(p => p.ThreadCount),5
Debugger.Sessions.First().Processes.Select(p => new { Name = p.Name, ThreadCount = p.Threads.Count()
}).OrderByDescending(p => p.ThreadCount),5

:
    [0x4]            :
        Name             : <Unknown Image>
        ThreadCount      : 0x73
    [0x708]          :
        Name             : explorer.exe
        ThreadCount      : 0x2d
    [0x37c]          :
        Name             : svchost.exe
        ThreadCount      : 0x2c
    [0x6b0]          :
        Name             : MsMpEng.exe
        ThreadCount      : 0x22
    [0x57c]          :
        Name             : svchost.exe
        ThreadCount      : 0x15
    [...]
```

This example shows the devices in the plug and play device tree grouped by the name of the physical device object's driver. Not all of the output is shown.

```
kd> dx -r2 Debugger.Sessions.First().Devices.DeviceTree.Flatten(n => n.Children).GroupBy(n =>
n.PhysicalDeviceObject->Driver->DriverName.ToDisplayString())
Debugger.Sessions.First().Devices.DeviceTree.Flatten(n => n.Children).GroupBy(n => n.PhysicalDeviceObject-
>Driver->DriverName.ToDisplayString())

:
    ["\"\\Driver\\PnpManager\""] :
        [0x0]            : HTREE\ROOT\0
        [0x1]            : ROOT\volmgr\0000 (volmgr)
        [0x2]            : ROOT\BasicDisplay\0000 (BasicDisplay)
        [0x3]            : ROOT\CompositeBus\0000 (CompositeBus)
        [0x4]            : ROOT\vdrvroot\0000 (vdrvroot)
         ...
```

Dx Command Tab Auto Completion

Contextual TAB key auto completion is aware of the LINQ query methods and will work for parameters of lambdas.

As an example, type (or copy and paste) the following text into the debugger. Then hit the TAB key several times

to cycle through potential completions.

```
dx -r2 Debugger.Sessions.First().Processes.Select(p => new {Name = p.Name, ThreadCount = p.Threads.Count()
}).OrderByDescending(p => p.
```

Press the TAB key until ".Name" appears. Add a closing parenthesis ")" and press enter to execute the command.

```
kd> dx -r2 Debugger.Sessions.First().Processes.Select(p => new {Name = p.Name, ThreadCount =
p.Threads.Count() }).OrderByDescending(p => p.Name)
Debugger.Sessions.First().Processes.Select(p => new {Name = p.Name, ThreadCount = p.Threads.Count()
}).OrderByDescending(p => p.Name) :
    [0x274]          :
        Name             : winlogon.exe
        ThreadCount      : 0x4
    [0x204]          :
        Name             : wininit.exe
        ThreadCount      : 0x2
    [0x6c4]          :
        Name             : taskhostex.exe
        ThreadCount      : 0x8
         ...
```

This example shows completion with a key comparator method. The substitution will show string methods, since the key is a string.

```
dx -r2 Debugger.Sessions.First().Processes.Select(p => new {Name = p.Name, ThreadCount = p.Threads.Count()
}).OrderByDescending(p => p.Name, (a, b) => a.
```

Press the TAB key until ".Length" appears. Add a closing parenthesis ")" and press enter to execute the command.

```
kd> dx -r2 Debugger.Sessions.First().Processes.Select(p => new {Name = p.Name, ThreadCount =
p.Threads.Count() }).OrderByDescending(p => p.Name, (a, b) => a.Length)
Debugger.Sessions.First().Processes.Select(p => new {Name = p.Name, ThreadCount = p.Threads.Count()
}).OrderByDescending(p => p.Name, (a, b) => a.Length) :
    [0x544]          :
        Name             : spoolsv.exe
        ThreadCount      : 0xc
    [0x4d4]          :
        Name             : svchost.exe
        ThreadCount      : 0xa
    [0x438]          :
        Name             : svchost.exe
```

User Defined Variables

A user defined variable can be defined by prefixing the variable name with @$. A user defined variable can be assigned to anything dx can utilize, for example, lambdas, the results of LINQ queries, etc.

You can create and set the value of a user variable like this.

```
kd> dx @$String1="Test String"
```

You can display the defined user variables using *Debugger.State.UserVariables* or *@$vars*.

```
kd> dx Debugger.State.UserVariables
Debugger.State.UserVariables :
    mySessionVar     :
    String1          : Test String
```

You can remove a variable using .Remove.

```
kd> dx @$vars.Remove("String1")
```

This example shows how to define a user variable to reference Debugger.Sesssions.

```
kd> dx @$mySessionVar = Debugger.Sessions
```

The user defined variable can then be used as shown below.

```
kd> dx -r2 @$mySessionVar
@$mySessionVar   :
    [0x0]            : Remote KD: KdSrv:Server=@{<Local>},Trans=@{COM:Port=\\.\com3,Baud=115200,Timeout=4000}
        Processes        :
        Devices
```

System Defined Variables

The following system defined variables can be used in any LINQ dx query.

- @$cursession - The current session
- @$curprocess - The current process
- @$curthread - The current thread

This example show the use of the system defined variables.

```
kd> dx @$curprocess.Threads.Count()
@$curprocess.Threads.Count() : 0x4
```

```
kd> dx -r1 @$curprocess.Threads
@$curprocess.Threads :
    [0x4adc]         :
    [0x1ee8]         :
    [0x51c8]         :
    [0x62d8]         :
     ...
```

User Defined Variables - Anonymous Types

This creation of dynamic objects is done using the C# anonymous type syntax (new { ... }). For more information see about anonymous types, see Anonymous Types (C# Programming Guide). This example create an anonymous type with an integer and string value.

```
kd> dx -r1 new { MyInt = 42, MyString = "Hello World" }
new { MyInt = 42, MyString = "Hello World" } :
    MyInt            : 42
    MyString         : Hello World
```

Function Objects (Lambda Expressions)

Many of the methods that are used to query data are based on the concept of repeatedly running a user provided function across objects in a collection. To support the ability to query and manipulate data in the debugger, the dx command supports lambda expressions using the equivalent C# syntax. A lambda expression is defined by usage of the => operator as follows:

(arguments) => (result)

To see how LINQ is used with dx, try this simple example to add together 5 and 7.

```
kd> dx ((x, y) => (x + y))(5, 7)
```

The dx command echos back the lambda expression and displays the result of 12.

```
((x, y) => (x + y))(5, 7)  : 12
```

This example lambda expression combines the strings "Hello" and "World".

```
kd> dx ((x, y) => (x + y))("Hello", "World")
((x, y) => (x + y))("Hello", "World") : HelloWorld
```

Supported LINQ Syntax - Query Methods

Any object which dx defines as iterable (be that a native array, a type which has NatVis written describing it as a container, or a debugger extension object) has a series of LINQ (or LINQ equivalent) methods projected onto it. Those query methods are described below. The signatures of the arguments to the query methods are listed after all of the query methods.

Filtering Methods

.Where (PredicateMethod)	Returns a new collection of objects containing every object in the input collection for which the predicate method returned true.

Projection Methods

.Flatten ([KeyProjectorMethod])	Takes an input container of containers (a tree) and flattens it into a single container which has every element in the tree. If the optional key projector method is supplied, the tree is considered a container of keys which are themselves containers and those keys are determined by a call to the projection method.

.Select (KeyProjectorMethod)	Returns a new collection of objects containing the result of calling the projector method on every object in the input collection.

Grouping Methods

.GroupBy (KeyProjectorMethod, [KeyComparatorMethod])	Returns a new collection of collections by grouping all objects in the input collection having the same key as determined by calling the key projector method. An optional comparator method can be provided.
Join (InnerCollection, Outer key selector method, Inner key selector method, Result selector method, [ComparatorMethod])	Joins two sequences based on key selector functions and extracts pairs of values. An optional comparator method can also be specified.
Intersect (InnerCollection, [ComparatorMethod])	Returns the set intersection, which means elements that appear in each of two collections. An optional comparator method can also be specified.
Union (InnerCollection, [ComparatorMethod])	Returns the set union, which means unique elements that appear in either of two collections. An optional comparator method can also be specified.

Data Set Methods

Contains (Object, [ComparatorMethod])	Determines whether a sequence contains a specified element. An optional comparator method can be provided that will be called each time the element is compared against an entry in the sequence.
Distinct ([ComparatorMethod])	Removes duplicate values from a collection. An optional comparator method can be provided to be called each time objects in the collection must be compared.
Except (InnerCollection, [ComparatorMethod])	Returns the set difference, which means the elements of one collection that do not appear in a second collection. An optional comparator method can be specified.
Concat (InnerCollection)	Concatenates two sequences to form one sequence.

Ordering Methods

.OrderBy (KeyProjectorMethod, [KeyComparatorMethod])	Sorts the collection in ascending order according to a key as provided by calling the key projection method on every object in the input collection. An optional comparator method can be provided.
.OrderByDescending (KeyProjectorMethod, [KeyComparatorMethod])	Sorts the collection in descending order according to a key as provided by calling the key projection method on every object in the input collection. An optional comparator method can be provided.

Aggregating Methods

Count ()	A method that returns the number of elements in the collection.
Sum ([ProjectionMethod])	Calculates the sum of the values in a collection. Can optionally specify a projector method to transform the elements before summation occurs.

Skip Methods

Skip (Count)	Skips elements up to a specified position in a sequence.
SkipWhile (PredicateMethod)	Skips elements based on a predicate function until an element does not satisfy the condition.

Take Methods

Take (Count)	Takes elements up to a specified position in a sequence.
TakeWhile (PredicateMethod)	Takes elements based on a predicate function until an element does not satisfy the condition.

Comparison Methods

SequenceEqual (InnerCollection, [ComparatorMethod])	Determines whether two sequences are equal by comparing elements in a pair-wise manner. An optional comparator can be specified.

Error Handling Methods

AllNonError (PredicateMethod)	Returns whether all non-error elements of a collection satisfy a given condition.
FirstNonError ([PredicateMethod])	Returns the first element of a collection that isn't an error.
LastNonError ([PredicateMethod])	Returns the last element of a collection that isn't an error.

Other Methods

.All (PredicateMethod)	Returns whether the result of calling the specified predicate method on every element in the input collection is true.
.Any (PredicateMethod)	Returns whether the result of calling the specified predicate method on any element in the input collection is true.

.First ([PredicateMethod])	Returns the first element in the collection. If the optional predicate is passed, returns the first element in the collection for which a call to the predicate returns true.
.Last ([PredicateMethod])	Returns the last element in the collection. If the optional predicate is passed, returns the last element in the collection for which a call to the predicate returns true.
Min([KeyProjectorMethod])	Returns the minimum element of the collection. An optional projector method can be specified to project each method before it is compared to others.
Max([KeyProjectorMethod])	Returns the maximum element of the collection. An optional projector method can be specified to project each method before it is compared to others.
Single([PredicateMethod])	Returns the only element from the list (or an error if the collection contains more than one element). If a predicate is specified, returns the single element that satisfies that predicate (if more than one element satisfies it, the function returns an error instead).

Signatures of the Arguments

KeyProjectorMethod : (obj => arbitrary key)	Takes an object of the collection and returns a key from that object.
KeyComparatorMethod: ((a, b) => integer value)	Takes two keys and compares them returning: -1 if (a < b) 0 if (a == b) 1 if (a > b)
PredicateMethod: (obj => boolean value)	Takes an object of the collection and returns true or false based on whether that object meets certain criteria.

Supported LINQ Syntax - String Manipulation

All string objects have the following methods projected into them, so that they are available for use:

Query Relevant Methods & Properties

.Contains (OtherString)	Returns a boolean value indicating whether the input string contains OtherString.
.EndsWith (OtherString)	Returns a boolean value indicating whether the input string ends with OtherString.
Length	A property which returns the length of the string.
.StartsWith (OtherString)	Returns a boolean value indicating whether the input string starts with OtherString.

.Substring (StartPos, [Length])	Returns a substring within the input string starting at the given starting position. If the optional length is supplied, the returned substring will be of the specified length; otherwise – it will go to the end of the string.

Miscellaneous Methods

.IndexOf (OtherString)	Returns the index of the first occurrence of OtherString within the input string.
.LastIndexOf (OtherString)	Returns the index of the last occurrence of OtherString within the input string.

Formatting Methods

.PadLeft (TotalWidth)	Adds spaces as necessary to the left side of the string in order to bring the total length of the string to the specified width.
.PadRight (TotalWidth)	Adds spaces as necessary to the right side of the string in order to bring the total length of the string to the specified width.
.Remove (StartPos, [Length])	Removes characters from the input string starting as the specified starting position. If the optional length parameter is supplied, that number of characters will be removed; otherwise – all characters to the end of the string will be removed.
.Replace (SearchString, ReplaceString)	Replaces every occurrence of SearchString within the input string with the specified ReplaceString.

String Object Projections

In addition to the methods which are projected directly onto string objects, any object which itself has a string conversion has the following method projected onto it, making it method available for use:

.ToDisplayString ( )	Returns a string conversion of the object. This is the string conversion which would be shown in a dx invocation for the object. You can provide a formatting specifier to format the output of ToDisplayString.

The following examples illustrate the use of format specifiers.

```
kd> dx (10).ToDisplayString("d")
(10).ToDisplayString("d") : 10

kd> dx (10).ToDisplayString("x")
(10).ToDisplayString("x") : 0xa

kd> dx (10).ToDisplayString("o")
(10).ToDisplayString("o") : 012

kd> dx (10).ToDisplayString("b")
(10).ToDisplayString("b")  : 0y1010
```

Debugging Plug and Play Example

This section illustrates how the built in debugger objects used with LINQ queries, can be used to debug plug and play objects.

View all devices

Use *Flatten* on the device tree to view all devices.

```
 1: kd> dx @$cursession.Devices.DeviceTree.Flatten(n => n.Children)
@$cursession.Devices.DeviceTree.Flatten(n => n.Children)
    [0x0]            : HTREE\ROOT\0
    [0x1]            : ROOT\volmgr\0000 (volmgr)
    [0x2]            : ROOT\BasicDisplay\0000 (BasicDisplay)
    [0x3]            : ROOT\CompositeBus\0000 (CompositeBus)
    [0x4]            : ROOT\vdrvroot\0000 (vdrvroot)
    [0x5]            : ROOT\spaceport\0000 (spaceport)
    [0x6]            : ROOT\KDNIC\0000 (kdnic)
    [0x7]            : ROOT\UMBUS\0000 (umbus)
    [0x8]            : ROOT\ACPI_HAL\0000
...
```

Grid Display

As with other dx commands, you can right click on a command after it was executed and click "Display as grid" or add "-g" to the command to get a grid view of the results.

```
# 0: kd> dx -g @$cursession.Devices.DeviceTree.Flatten(n => n.Children)
================================================================================================================================
================================================================================================================================
=============================================================================================
# =                                                          = (+) DeviceNodeObject = InstancePath
= ServiceName               = (+) PhysicalDeviceObject                                    = State
= (+) Resoures = (+) Children       =
================================================================================================================================
================================================================================================================================
=============================================================================================
= [0x0] : HTREE\ROOT\0                                        - {...}                - HTREE\ROOT\0
-                          - 0xffffb6075614be40 : Device for "\Driver\PnpManager"        - DeviceNodeStarted
(776)        - {...}        - [object Object]    =
= [0x1] : ROOT\volmgr\0000 (volmgr)                           - {...}                - ROOT\volmgr\0000
- volmgr                   - 0xffffb607561fbe40 : Device for "\Driver\PnpManager"        - DeviceNodeStarted
(776)        - {...}        - [object Object]    =
= [0x2] : ROOT\BasicDisplay\0000 (BasicDisplay)               - {...}                -
ROOT\BasicDisplay\0000                                     - BasicDisplay              - 0xffffb607560739b0
: Device for "\Driver\PnpManager"        - DeviceNodeStarted (776)        - {...}        - [object Object]
=
= [0x3] : ROOT\CompositeBus\0000 (CompositeBus)               - {...}                -
ROOT\CompositeBus\0000                                     - CompositeBus              - 0xffffb607561f9060
: Device for "\Driver\PnpManager"        - DeviceNodeStarted (776)        - {...}        - [object Object]
=
...
```

View Devices by State

Use *Where* to specify a specific device state.

```
dx @$cursession.Devices.DeviceTree.Flatten(n => n.Children).Where(n => n.State <operator> <state number>)
```

For example to view devices in state DeviceNodeStarted use this command.

```
1: kd>  dx @$cursession.Devices.DeviceTree.Flatten(n => n.Children).Where(n => n.State == 776)
@$cursession.Devices.DeviceTree.Flatten(n => n.Children).Where(n => n.State == 776)
    [0x0]            : HTREE\ROOT\0
    [0x1]            : ROOT\volmgr\0000 (volmgr)
    [0x2]            : ROOT\BasicDisplay\0000 (BasicDisplay)
    [0x3]            : ROOT\CompositeBus\0000 (CompositeBus)
    [0x4]            : ROOT\vdrvroot\0000 (vdrvroot)
...
```

View Not Started Devices

Use this command to view devices not in state DeviceNodeStarted.

```
1: kd>  dx @$cursession.Devices.DeviceTree.Flatten(n => n.Children).Where(n => n.State != 776)
@$cursession.Devices.DeviceTree.Flatten(n => n.Children).Where(n => n.State != 776)
    [0x0]            : ACPI\PNP0C01\1
    [0x1]            : ACPI\PNP0000\4&215d0f95&0
    [0x2]            : ACPI\PNP0200\4&215d0f95&0
    [0x3]            : ACPI\PNP0100\4&215d0f95&0
    [0x4]            : ACPI\PNP0800\4&215d0f95&0
    [0x5]            : ACPI\PNP0C04\4&215d0f95&0
    [0x6]            : ACPI\PNP0700\4&215d0f95&0 (fdc)
    [0x7]            : ACPI\PNP0C02\1
    [0x8]            : ACPI\PNP0C02\2
```

View Devices by Problem Code

Use the *DeviceNodeObject.Problem* object to view devices that have specific problem codes.

```
dx @$cursession.Devices.DeviceTree.Flatten(n => n.Children).Where(n => n.DeviceNodeObject.Problem <operator>
<problemCode>)
```

For example, to view devices that have a non zero problem code use this command. This provides similar information to "**!devnode** 0 21".

```
1: kd> dx @$cursession.Devices.DeviceTree.Flatten(n => n.Children).Where(n => n.DeviceNodeObject.Problem !=
0)
@$cursession.Devices.DeviceTree.Flatten(n => n.Children).Where(n => n.DeviceNodeObject.Problem != 0)
    [0x0]            : HTREE\ROOT\0
    [0x1]            : ACPI\PNP0700\4&215d0f95&0 (fdc)
```

View All Devices Without a Problem

Use this command to view all devices without a problem

```
1: kd> dx @$cursession.Devices.DeviceTree.Flatten(n => n.Children).Where(n => n.DeviceNodeObject.Problem ==
0)
@$cursession.Devices.DeviceTree.Flatten(n => n.Children).Where(n => n.DeviceNodeObject.Problem == 0)
    [0x0]            : ROOT\volmgr\0000 (volmgr)
    [0x1]            : ROOT\BasicDisplay\0000 (BasicDisplay)
    [0x2]            : ROOT\CompositeBus\0000 (CompositeBus)
    [0x3]            : ROOT\vdrvroot\0000 (vdrvroot)
...
```

View All Devices With a Specific Problem

Use this command to view devices with a problem state of 0x16.

```
1: kd> dx @$cursession.Devices.DeviceTree.Flatten(n => n.Children).Where(n => n.DeviceNodeObject.Problem ==
0x16)
@$cursession.Devices.DeviceTree.Flatten(n => n.Children).Where(n => n.DeviceNodeObject.Problem == 0x16)
    [0x0]            : HTREE\ROOT\0
    [0x1]            : ACPI\PNP0700\4&215d0f95&0 (fdc)
```

View Devices by Function Driver

Use this command to view devices by function driver.

```
dx @$cursession.Devices.DeviceTree.Flatten(n => n.Children).Where(n => n.ServiceName <operator> <service
name>)
```

To view devices using a certain function driver, such as atapi, use this command.

```
1: kd> dx @$cursession.Devices.DeviceTree.Flatten(n => n.Children).Where(n => n.ServiceName == "atapi")
@$cursession.Devices.DeviceTree.Flatten(n => n.Children).Where(n => n.ServiceName == "atapi")
    [0x0]            : PCIIDE\IDEChannel\4&10bf2f88&0&0 (atapi)
    [0x1]            : PCIIDE\IDEChannel\4&10bf2f88&0&1 (atapi)
```

Viewing a List of Boot Start Drivers

To view the list of what winload loaded as boot start drivers, you need to be in a context where you have access to

the LoaderBlock and early enough the LoaderBlock is still around. For example, during nt!IopInitializeBootDrivers. A breakpoint can be set to stop in this context.

```
1: kd> g
Breakpoint 0 hit
nt!IopInitializeBootDrivers:
8225c634 8bff            mov     edi,edi
```

Use the ?? command to display the boot driver structure.

```
1: kd> ?? LoaderBlock->BootDriverListHead
struct _LIST_ENTRY
 [ 0x808c9960 - 0x808c8728 ]
   +0x000 Flink            : 0x808c9960 _LIST_ENTRY [ 0x808c93e8 - 0x808a2e18 ]
   +0x004 Blink            : 0x808c8728 _LIST_ENTRY [ 0x808a2e18 - 0x808c8de0 ]
```

Use the Debugger.Utility.Collections.FromListEntry debugger object to view of the data, using the starting address of the nt!_LIST_ENTRY structure.

```
1: kd> dx Debugger.Utility.Collections.FromListEntry(*(nt!_LIST_ENTRY *)0x808c9960,
"nt!_BOOT_DRIVER_LIST_ENTRY", "Link")
Debugger.Utility.Collections.FromListEntry(*(nt!_LIST_ENTRY *)0x808c9960, "nt!_BOOT_DRIVER_LIST_ENTRY",
"Link")
    [0x0]            [Type: _BOOT_DRIVER_LIST_ENTRY]
    [0x1]            [Type: _BOOT_DRIVER_LIST_ENTRY]
    [0x2]            [Type: _BOOT_DRIVER_LIST_ENTRY]
    [0x3]            [Type: _BOOT_DRIVER_LIST_ENTRY]
    [0x4]            [Type: _BOOT_DRIVER_LIST_ENTRY]
    [0x5]            [Type: _BOOT_DRIVER_LIST_ENTRY]
...
```

Use the -g option to create a grid view of the data.

```
dx -r1 -g Debugger.Utility.Collections.FromListEntry(*(nt!_LIST_ENTRY *)0x808c9960,
"nt!_BOOT_DRIVER_LIST_ENTRY", "Link")
```

View devices by Capability

View devices by capability using the DeviceNodeObject.CapabilityFlags object.

```
dx -r1 @$cursession.Devices.DeviceTree.Flatten(n => n.Children).Where(n =>
(n.DeviceNodeObject.CapabilityFlags & <flag>) != 0)
```

This table summarizes the use of the dx command with common device capability flags.

Removable	`dbgcmd 0: kd> dx -r1 @$cursession.Devices.DeviceTree.Flatten(n => n.Children).Where(n => (n.DeviceNodeObject.CapabilityFlags & 0x10) != 0) @$cursession.Devices.DeviceTree.Flatten(n => n.Children).Where(n => (n.DeviceNodeObject.CapabilityFlags & 0x10) != 0) [0x0] : SWD\PRINTENUM{2F8DBBB6-F246-4D84-BB1D-AA8761353885} [0x1] : SWD\PRINTENUM{F210BC77-55A1-4FCA-AA80-013E2B408378} [0x2] : SWD\PRINTENUM{07940A8E-11F4-46C3-B714-7FF9B87738F8} [0x3] : DISPLAY\Default_Monitor\6&1a097cd8&0&UID5527112 (monitor)`
UniqueID	`dbgcmd 0: kd> dx -r1 @$cursession.Devices.DeviceTree.Flatten(n => n.Children).Where(n => (n.DeviceNodeObject.CapabilityFlags & 0x40) != 0) @$cursession.Devices.DeviceTree.Flatten(n => n.Children).Where(n => (n.DeviceNodeObject.CapabilityFlags & 0x40) != 0) [0x0] : HTREE\ROOT\0 [0x1] : ROOT\volmgr\0000 (volmgr) [0x2] : ROOT\spaceport\0000 (spaceport) ...`
SilentInstall	`dbgcmd 0: kd> dx -r1 @$cursession.Devices.DeviceTree.Flatten(n => n.Children).Where(n => (n.DeviceNodeObject.CapabilityFlags & 0x80) != 0) @$cursession.Devices.DeviceTree.Flatten(n => n.Children).Where(n => (n.DeviceNodeObject.CapabilityFlags & 0x80) != 0) [0x0] : HTREE\ROOT\0 [0x1] : ROOT\volmgr\0000 (volmgr) [0x2] : ROOT\spaceport\0000 (spaceport) ...`
RawDeviceOk	`dbgcmd 0: kd> dx -r1 @$cursession.Devices.DeviceTree.Flatten(n => n.Children).Where(n => (n.DeviceNodeObject.CapabilityFlags & 0x100) != 0) @$cursession.Devices.DeviceTree.Flatten(n => n.Children).Where(n => (n.DeviceNodeObject.CapabilityFlags & 0x100) != 0) [0x0] : HTREE\ROOT\0 [0x1] : SWD\MMDEVAPI\MicrosoftGSWavetableSynth [0x2] : SWD\IP_TUNNEL_VBUS\IP_TUNNEL_DEVICE_ROOT ...`
SurpriseRemovalOK	`dbgcmd 0: kd> dx -r1 @$cursession.Devices.DeviceTree.Flatten(n => n.Children).Where(n => (n.DeviceNodeObject.CapabilityFlags & 0x200) != 0) @$cursession.Devices.DeviceTree.Flatten(n => n.Children).Where(n => (n.DeviceNodeObject.CapabilityFlags & 0x200) != 0) [0x0] : SWD\MMDEVAPI\MicrosoftGSWavetableSynth [0x1] : SWD\IP_TUNNEL_VBUS\IP_TUNNEL_DEVICE_ROOT [0x2] : SWD\PRINTENUM\PrintQueues ...`

For more information about the CapabilityFlags, see **DEVICE_CAPABILITIES**.

See also

dx (Display Debugger Object Model Expression)

Native Debugger Objects in NatVis

Native Debugger Objects in JavaScript Extensions

Debugger Data Model Function Aliases

12/5/2018 • 5 minutes to read • Edit Online

Function aliases are a unique short name by which a user of the debugger can access functionality defined in a debugger extension (whether written in C++ or some scripting environment such as JavaScript). This short name gets associated with a data model function object (an object which implements IModelMethod). That function takes an arbitrary number of arguments and returns a single value. The effect of invoking a function alias and what is done with that value depend on how the function alias is invoked and what host debugger it is invoked within.

This topic assumes the reader is familiar with the debugger object model and JavaScript. For information about using debugger objects with JavaScript, see Native Debugger Objects in JavaScript Extensions.

Some of the examples shown here use the dx command, for more information about working with the dx command, see dx (Display Debugger Object Model Expression). In addition the LINQ is used which is described in Using LINQ With the debugger objects.

Using function alias as extension commands

All function aliases created in WinDbg Preview can be invoked as if they were debug extension **!** "bang" commands. If the function takes no arguments, simply invoking !aliasName will cause the function to be called and the result value to be displayed. As an example (created with JavaScript extensibility)

As an example, this function provides two constant values, *pi* and *e*.

```
function __constants()
{
    return { math_constants : { pi :  3.1415926535 , e :  2.7182818284}, description : "Archimedes' pi and
Euler's number e" };
}

function initializeScript()
{
    return [new host.functionAlias(__constants, "constants")];
}
```

The results of calling the function alias are shown here.

```
0:000> !constants
@$constants()                : [object Object]
    math_constants   : [object Object]
    description      : Archimedes' pi and Euler's number e
```

DML links to complex objects will be generated automatically. Clicking the math_constants shown above will result in the following output.

```
0:000> dx -r1 @$constants().math_constants
@$constants().math_constants                 : [object Object]
    pi               : 3.141593
    e                : 2.718282
```

If the function has arguments, they can be supplied after the function alias command itself. Note that whatever comes after the function alias command is considered an expression and is evaluated as such. The text string is not

passed directly to the function. For a single argument expression, it can come after the function alias command itself. For multiple arguments, they should be parenthesized as if it were a function call as illustrated in this next example.

```
function __oneArgument(x)
{
    return -x;
}

function __twoArguments(x, y)
{
    return x + y;
}

function initializeScript()
{
    return [new host.functionAlias(__oneArgument, "neg"),
            new host.functionAlias(__twoArguments, "add")];
}
```

These two functions can be called as shown here.

```
0:000> !neg 42
@$neg(42)        : -42

0:000> !add (5, 7)
@$add(5, 7)      : 0xc
```

Function alias use with the dx expression evaluator

In addition to the debug extension **!** "bang" command syntax for invoking an aliased function, all of the names associated with function aliases are directly available in the dx expression evaluator when prefixed by *@$* as shown here.

```
0:000> dx @$neg(42)
@$neg(42)        : -42

0:000> dx @$add(99, 77)
@$add(99, 77)    : 0xb0
```

Function alias design considerations

A function alias should never be the sole way in which functionality in the vast majority of data model extensions is exposed. A data model extension (be it in C++ or JavaScript) should nearly always include the data it is exposing associated with a type or other debugger concept. Things associated with processes should be under Debugger.Models.Process or a sub-namespace of that object. A function alias can be a convenient way for getting to (or transforming) data that might require a significantly longer query.

As a kernel mode example, the following query takes the PnP device tree and flattens it into a simple flat list of devices:

```
0: kd> dx @$cursession.Devices.DeviceTree.Flatten(n => n.Children),5
@$cursession.Devices.DeviceTree.Flatten(n => n.Children),5
    [0x0]            : HTREE\ROOT\0
    [0x1]            : ROOT\volmgr\0000 (volmgr)
    [0x2]            : ROOT\BasicDisplay\0000 (BasicDisplay)
    [0x3]            : ROOT\CompositeBus\0000 (CompositeBus)
    [0x4]            : ROOT\vdrvroot\0000 (vdrvroot)
    [...]
```

This JavaScript code shows how this can be implemented as a function alias.

```
function __flatDevices()
{
    return host.currentSession.Devices.DeviceTree.Flatten(n => n.Children);
}

function initializeScript()
{
    return [new host.functionAlias(__flatDevices, "devices")];
}
```

The function alias can then be invoked as a debug extension command.

```
0: kd> !devices
@$devices()
    [0x0]            : HTREE\ROOT\0
    [0x1]            : ROOT\volmgr\0000 (volmgr)
    [0x2]            : ROOT\BasicDisplay\0000 (BasicDisplay)
    [0x3]            : ROOT\CompositeBus\0000 (CompositeBus)
    [0x4]            : ROOT\vdrvroot\0000 (vdrvroot)
    [0x5]            : ROOT\spaceport\0000 (spaceport)

    ...
```

One of the advantages with using a function alias, is that it can be further refined using the dx syntax. In this example, a where clause is added to look for device nodes that contain "Harddisk".

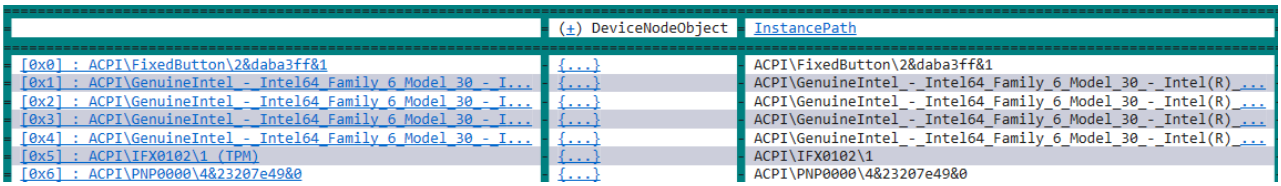
```
0: kd> dx @$devices().Where(n => n.InstancePath.Contains("Harddisk"))
@$devices().Where(n => n.InstancePath.Contains("Harddisk"))
    [0x0]            : STORAGE\VolumeSnapshot\HarddiskVolumeSnapshot1
    [0x1]            : STORAGE\VolumeSnapshot\HarddiskVolumeSnapshot2
    [0x2]            : STORAGE\VolumeSnapshot\HarddiskVolumeSnapshot3
    [0x3]            : STORAGE\VolumeSnapshot\HarddiskVolumeSnapshot4
    [0x4]            : STORAGE\VolumeSnapshot\HarddiskVolumeSnapshot5
    [0x5]            : STORAGE\VolumeSnapshot\HarddiskVolumeSnapshot6
```

LINQ commands such as the following can be used with functional aliases - .All, .Any, .Count, .First, .Flatten, .GroupBy, .Last, .OrderBy, .OrderByDescending, .Select, and .Where. These methods follow (as closely as possible) the C# LINQ method form. For more information see Using LINQ With the debugger objects.

Grid Display

As with other dx commands, you can right click on a command after it was executed and click "Display as grid" or add "-g" to the command to get a grid view of the results. You can then click on any column to sort, for example on InstancePath.

```
0: kd> dx -g @$devices().OrderBy(obj => obj.@"InstancePath")
```



	(+) DeviceNodeObject	InstancePath
[0x0] : ACPI\FixedButton\2&daba3ff&1	{...}	ACPI\FixedButton\2&daba3ff&1
[0x1] : ACPI\GenuineIntel_-_Intel64_Family_6_Model_30_-_I...	{...}	ACPI\GenuineIntel_-_Intel64_Family_6_Model_30_-_Intel(R)_...
[0x2] : ACPI\GenuineIntel_-_Intel64_Family_6_Model_30_-_I...	{...}	ACPI\GenuineIntel_-_Intel64_Family_6_Model_30_-_Intel(R)_...
[0x3] : ACPI\GenuineIntel_-_Intel64_Family_6_Model_30_-_I...	{...}	ACPI\GenuineIntel_-_Intel64_Family_6_Model_30_-_Intel(R)_...
[0x4] : ACPI\GenuineIntel_-_Intel64_Family_6_Model_30_-_I...	{...}	ACPI\GenuineIntel_-_Intel64_Family_6_Model_30_-_Intel(R)_...
[0x5] : ACPI\IFX0102\1 (TPM)	{...}	ACPI\IFX0102\1
[0x6] : ACPI\PNP0000\4&23207e49&0	{...}	ACPI\PNP0000\4&23207e49&0

Process Threads Example

Debugger objects are projected into a namespace rooted at "Debugger". Processes, modules, threads, stacks, stack frames, and local variables are all available to be used in a LINQ query.

This example JavaScript shows how to display the thread count for the current sessions processes:

```
function __Processes()
{
    return host.currentSession.Processes.Select(p => ({Name: p.Name, ThreadCount: p.Threads.Count()}));
}

function initializeScript()
{
    return [new host.functionAlias(__Processes, "Processes")];
}
```

This shows the example output with the !Processes function alias.

```
0: kd> !Processes
@$Processes()
    [0x0]            : [object Object]
    [0x4]            : [object Object]
    [0x1b4]          : [object Object]
    [0x248]          : [object Object]
    [0x2c4]          : [object Object]
    [0x340]          : [object Object]
    [0x350]          : [object Object]
    [0x3d4]          : [object Object]
    [0x3e8]          : [object Object]
    [0x4c]           : [object Object]
    [0x214]          : [object Object]
    [0x41c]          : [object Object]
    [0x494]          : [object Object]

...
```

In this example the top 5 process with the largest thread count are displayed.

```
0: kd> dx -r1 @$Processes().OrderByDescending(p =>p.ThreadCount),5
@$Processes().OrderByDescending(p =>p.ThreadCount),5
    [0x4]            : [object Object]
    [0x180]          : [object Object]
    [0x978]          : [object Object]
    [0xda4]          : [object Object]
    [0x3e8]          : [object Object]
    [...]
```

See also

dx (Display Debugger Object Model Expression)

Using LINQ With the debugger objects

Native Debugger Objects in NatVis

Native Debugger Objects in JavaScript Extensions

Debugger Data Model C++ Overview

10/23/2019 • 8 minutes to read • Edit Online

This topic provides an overview of how to use Debugger Data Model C++ Interfaces to extend and customize the capabilities of the debugger.

This topic is part of a series which describes the interfaces accessible from C++, how to use them to build a C++ based debugger extension, and how to make use of other data model constructs (e.g.: JavaScript or NatVis) from a C++ data model extension.

Debugger Data Model C++ Overview

Debugger Data Model C++ Interfaces

Debugger Data Model C++ Objects

Debugger Data Model C++ Additional Interfaces

Debugger Data Model C++ Concepts

Debugger Data Model C++ Scripting

Overview of the Debugger Data Model C++ Interface

The debugger data model is an extensible object model that is central to the way in which new debugger extensions (including those in JavaScript, NatVis, and C++) both consume information from the debugger and produce information that can be accessed from the debugger as well as other extensions. Constructs which are written to the data model APIs are available in the debugger's newer (dx) expression evaluator as well as from JavaScript extensions or C++ extensions.

To illustrate the goals of the debugger data model, consider this traditional debugger command.

```
0: kd> !process 0 0
PROCESS ffffe0007e6a7780
    SessionId: 1  Cid: 0f68    Peb: 7ff7cfe7a000  ParentCid: 0f34
    DirBase: 1f7fb9000  ObjectTable: ffffc001cec82780  HandleCount:  34.
    Image: echoapp.exe
...
```

The debugger command is using a binary mask and it provides text only output in non-standard ways. The text output is difficult to consume, format, or extend and the layout is specific to this command.

Contrast this to the debugger data model dx (Display Debugger Object Model Expression) command.

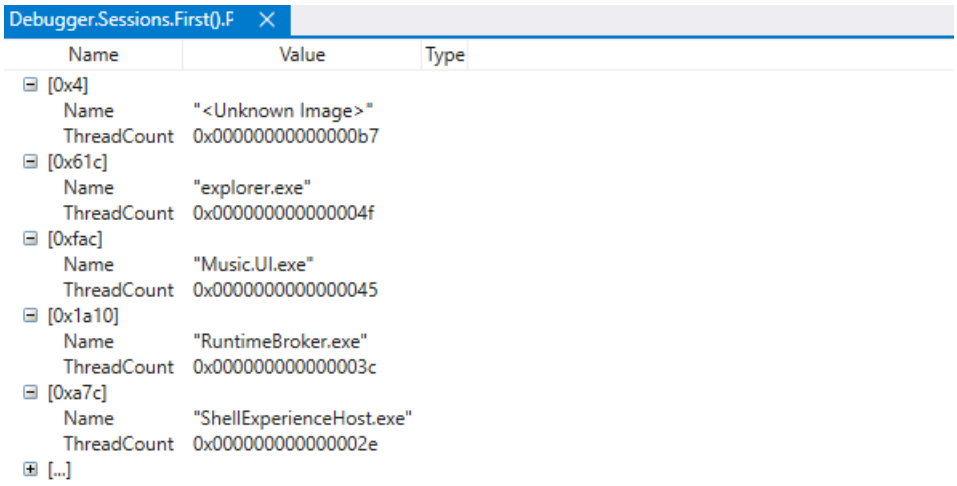
```
dx @$cursession.Processes.Where(p => p.Threads.Count() > 5)
```

This command uses a standard data model that is discoverable, extensible and composable in uniform ways.

Logically name spacing things and extending on specific objects allows for the discovery of debugger extension functionality.

> **TIP**
>
> Because the Data Model C++ Object interfaces can be very verbose to implement a full C++ helper library for the data model which uses a full C++ exception and template programming paradigm is recommended. For more information, see Using the DbgModelClientEx Library later in this topic.

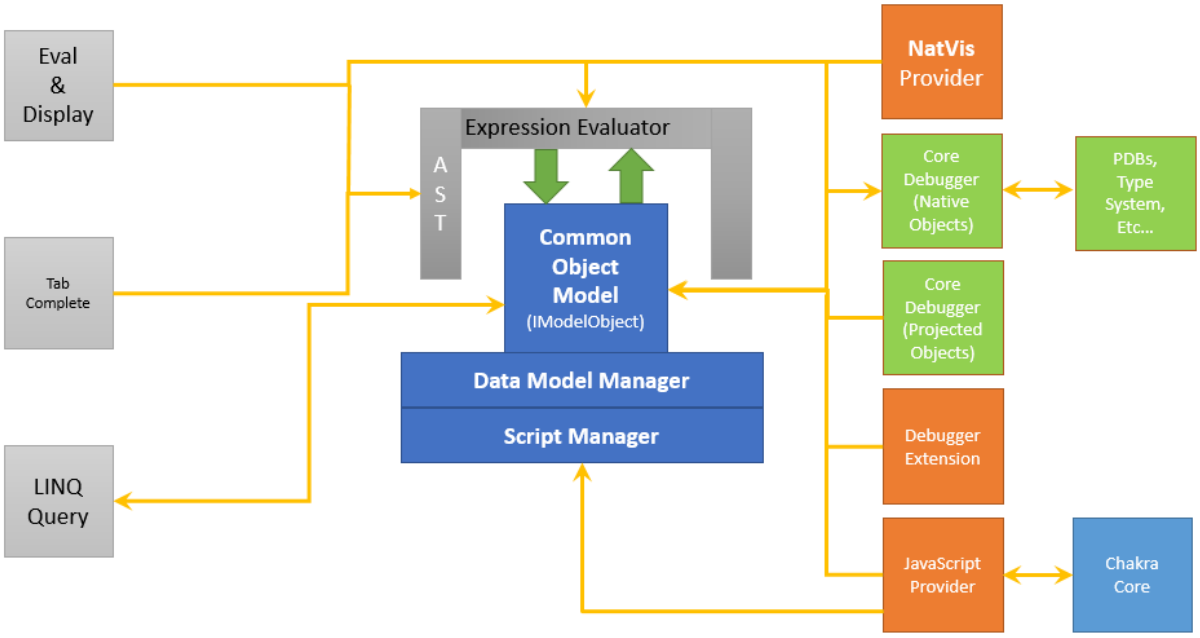
The data model is the way that the new WinDbg Preview debugger, shows most things. Many elements in the new UI can be queried, extended, or scripted, because they are powered by the data model. For more information, see WinDbg Preview - Data Model.



Data Model Architectural View

The following diagram summarizes the major elements of the debugger data model architecture.

- To the left side, UI elements are shown that provide access to the objects and support such functionality as LINQ queries.
- On the right side of the diagram are components that provide data to debugger data model. This includes custom NatVis, JavaScript and C++ debugger data model extensions.



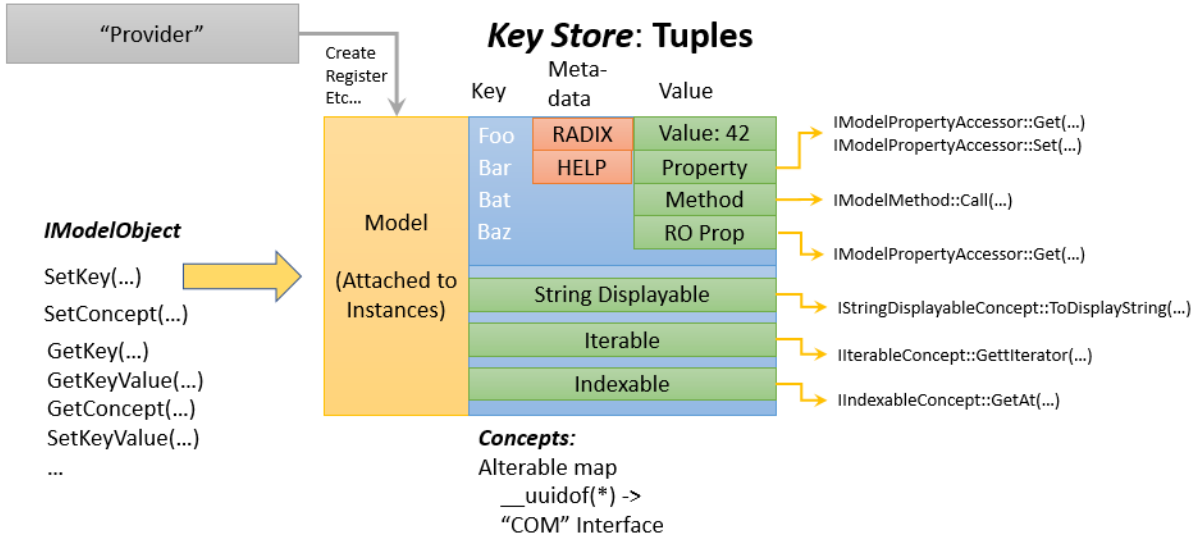
Object Model

At the center of the Debugger Data Model is a uniform object representation in which everything is an instance of the IModelObject interface. While such an object may represent an intrinsic (e.g.: an integer value) or another data

model interface, it often represents a dynamic object – a dictionary of key/value/metadata tuples and a set of concepts which describe abstract behaviors.

This diagram shows how the IModelObject uses Key Stores to contain values that a provider can create, register and manipulate.

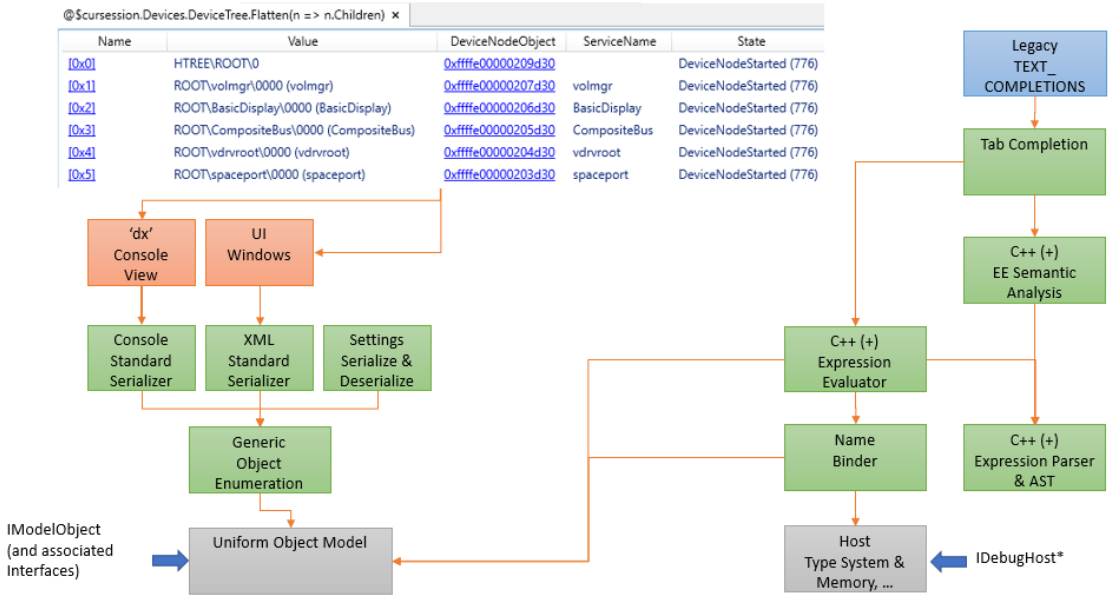
- It shows a *provider*, that provides information to the object model
- On the left it shows the *IModelObject*, that is the common object model that is used to manipulate objects.
- In the center is the *Key Store* that is used to store and access values.
- At the bottom it shows *Concepts* that support objects with functionality such as the ability to convert to a displayable string or be indexed.



The Data Model: A Consumer View

The next diagram shows a consumer view of the data model. In the example the dx (Display Debugger Object Model Expression) command is being used to query information.

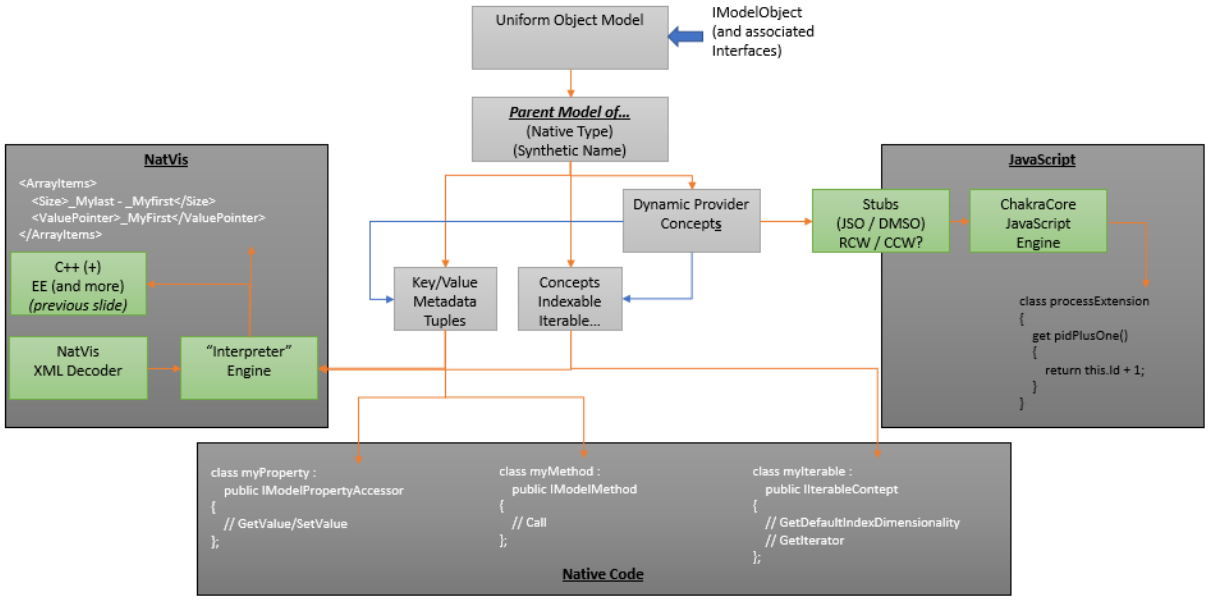
- The Dx command communicates through a serializer to the object enumeration interface.
- IDebugHost* objects are used to gather information from the debugger engine.
- Expression and semantic evaluators are used to send the request to the debugger engine.



The Data Model: A Producer View

This diagram shows a producer view of the data model.

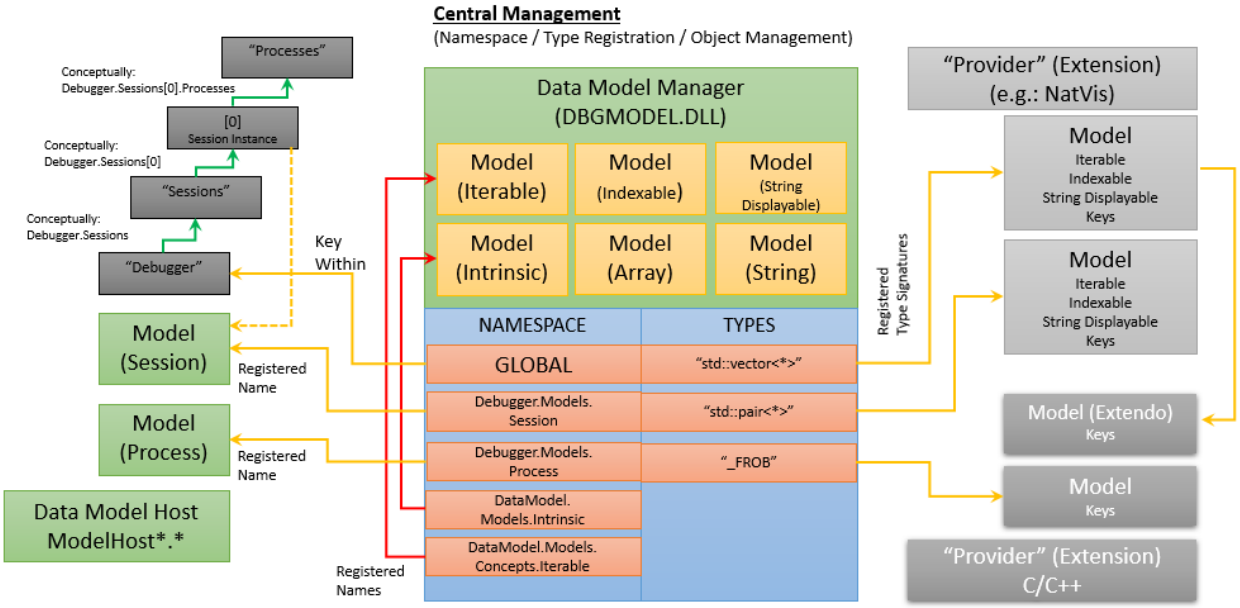
- A NatVis provider is shown on the left that consumes XML that defines additional functionality.
- A JavaScript provider can take advantage of *Dynamic Provider Concepts* to manipulate information in real time.
- The bottom shows a native code provider that can also define additional functionality.



Data Model Manager

This diagram shows the central role that the data model manager plays in the management of objects.

- The Data Model Manager acts as a central registrar for all objects.
- On the left it shows how standard debugger elements such as sessions and process are registered.
- The namespace block shows the central registration list.
- The right side of the diagram shows two providers, one for NatVis on the top, and a C/C++ extension on the bottom.



Summary of Debugger Data Model Interfaces

There are a multitude of C++ interfaces which comprise different pieces of the data model. In order to approach these interfaces in a consistent and easy manner, they are broken down by general category. The main areas here:

The General Object Model

The first and most important set of interfaces define how to get access to the core data model and how to access and manipulate objects. IModelObject is the interface which represents every object in the data model (much like C#'s object). This is the main interface of interest for both consumers of and producers to the data model. The other interfaces are mechanisms for accessing different aspects of objects. The following interfaces are defined for this category:

Bridges Between DbgEng and the Data Model

IHostDataModelAccess

Main Interfaces

IModelObject

IKeyStore

IModelIterator

IModelPropertyAccessor

IModelMethod

IKeyEnumerator

IRawEnumerator

IModelKeyReference / IModelKeyReference2

Concept Interfaces

IStringDisplayableConcept

IIterableConcept

IIndexableConcept

IPreferredRuntimeTypeConcept

IDataModelConcept

IDynamicKeyProviderConcept

IDynamicConceptProviderConcept

Management of Data Models and Extensibility

The Data Model Manager is the core component which manages how all extensibility occurs. It is the central repository of a set of tables which map both native types to extension points as well as synthetic constructs to extension points. In addition, it is the entity which is responsible for the boxing of objects (conversion of ordinal values or strings into IModelObject's).

The following interfaces are defined for this category:

General Data Model Manager Access

IDataModelManager / IDataModelManager2

Script Management

IDataModelScriptManager

IDataModelScriptProviderEnumerator

Access to the Debugger's Type System and Memory Spaces

The underlying type system and memory spaces of the debugger are exposed in detail for extensions to make use of. The following interfaces are defined for this category:

General Host (Debugger) Interfaces

IDebugHost

IDebugHostStatus

IDebugHostContext

IDebugHostMemory / IDebugHostMemory2

IDebugHostErrorSink

IDebugHostEvaluator / IDebugHostEvaluator2

IDebugHostExtensibility

Host (Debugger) Type System Interfaces

IDebugHostSymbols

IDebugHostSymbol / IDebugHostSymbol2

IDebugHostModule

IDebugHostType / IDebugHostType2

IDebugHostConstant

IDebugHostField

IDebugHostData

IDebugHostBaseClass IDebugHostPublic

IDebugHostModuleSignature

IDebugHostTypeSignature

Host (Debugger) Support for Scripting

IDebugHostScriptHost

Authoring and Consuming Scripts

The Data Model also has a general notion of what a script is and how to debug one. It is entirely possible for a debugger extension to come along and define a general bridge between the data model and another dynamic language (usually a scripting environment). This set of interfaces is how that is accomplished as well as how a debugger UI can make use of such scripts.

The following interfaces are defined for this category:

General Script Interfaces

IDataModelScriptProvider

IDataModelScript

IDataModelScriptClient

IDataModelScriptHostContext

IDataModelScriptTemplate

IDataModelScriptTemplateEnumerator

IDataModelNameBinder

Script Debugger Interfaces

IDataModelScriptDebug

IDataModelScriptDebugClient

IDataModelScriptDebugStack

IDataModelScriptDebugStackFrame

IDataModelScriptDebugVariableSetEnumerator

IDataModelScriptDebugBreakpoint

IDataModelScriptDebugBreakpointEnumerator

Using the DbgModelClientEx Library

Overview

The Data Model C++ Object Interfaces to the data model can be very verbose to implement. While they allow for full manipulation of the data model, they require implementation of a number of small interfaces to extend the data model (e.g.: an IModelPropertyAccessor implementation for each dynamic fetchable property which is added). In addition to this, the HRESULT based programming model adds a significant amount of boiler plate code that is used for error checking.

In order to minimize some of this work, there is a full C++ helper library for the data model which uses a full C++ exception and template programming paradigm. Use of this library allows for more concise code when consuming or extending the data model and is recommended.

There are two important namespaces in the helper library:

Debugger::DataModel::ClientEx - helpers for consumption of the data model

Debugger::DataModel::ProviderEx - helpers for extension of the data model

For additional information on using the DbgModelClientEx library, see the readme file at this github site:

https://github.com/Microsoft/WinDbg-Libraries/tree/master/DbgModelCppLib

HelloWorld C++ Sample

To see how the DbgModelClientEx library can be used, review the Data Model HelloWorld C++ sample here.

https://github.com/Microsoft/WinDbg-Samples/tree/master/DataModelHelloWorld

The sample includes:

- HelloProvider.cpp - This is an implementation of a provider class which adds a new example property "Hello" to the debugger's notion of a process.
- SimpleIntroExtension.cpp - This is a simple debugger extension which adds a new example property "Hello" to the debugger's notion of a process. This extension is written against the Data Model C++17

Helper Library. It is far preferable to write extensions against this library rather than the raw COM ABI due to the volume (and complexity) of glue code which is required.

JavaScript and COM Samples

In order to better understand the varying ways to write a debugger extension with the data model, there are three versions of the data model HelloWorld extension available here:

https://github.com/Microsoft/WinDbg-Samples/tree/master/DataModelHelloWorld

- JavaScript - A version written in JavaScript
- C++17 - A version written against the Data Model C++17 Client Library
- COM - A version written against the raw COM ABI (only using WRL for COM helpers)

Related topics

Debugger Data Model C++ Interfaces

Debugger Data Model C++ Objects

Debugger Data Model C++ Additional Interfaces

Debugger Data Model C++ Concepts

Debugger Data Model C++ Scripting

Debugger Data Model C++ Interfaces

10/23/2019 • 40 minutes to read • Edit Online

This topic provides and overview of how to use Debugger Data Model C++ Interfaces to extend and customize the capabilities of the debugger.

Debugger Data Model C++ Host Interfaces

The Debugger Data Model Host

The Debugger Data Model is designed to be a componentized system which can be hosted in a variety of different contexts. Normally, the Data Model is hosted in the context of a debugger application. In order to be a host of the data model, a number of interfaces need to be implemented to expose core aspects of the debugger: its targeting, its memory spaces, its evaluator, its symbolic and type system, etc... While these interfaces are implemented by any application wishing to host the data model, they are consumed by both the core data model as well as any extension which interoperates with the data model.

The set of core interfaces are:

INTERFACE NAME	DESCRIPTION
IDebugHost	The core interface to the debug host.
IDebugHostStatus	An interface allowing a client to query for the status of the host.
IDebugHostContext	An abstraction of a context within the host (e.g.: a particular target, a particular process, a particular address space, etc...)
IDebugHostErrorSink	An interface implemented by callers to receive errors from certain portions of the host and data model
IDebugHostEvaluator / IDebugHostEvaluator2	The debug host's expression evaluator.
IDebugHostExtensibility	An interface for extending the capabilities of the host or portions of it (such as the expression evaluator).

The type system and symbolic interfaces are:

INTERFACENAME	DESCRIPTION
IDebugHostSymbols	Core interface which provides access to and resolution of symbols
IDebugHostSymbol / IDebugHostSymbol2	Represents a single symbol of any kind. The particular symbol is a derivation of this interface.
IDebugHostModule	Represents a module loaded within a process. This is a kind of symbol.
IDebugHostType / IDebugHostType2	Represents a native/language type.

INTERFACENAME	DESCRIPTION
IDebugHostConstant	Represents a constant within symbolic information (e.g.: a non-type template argument in C++)
IDebugHostField	Represents a field within a structure or class.
IDebugHostData	Represents data within a module (were this within a structure or class it would be an IDebugHostField)
IDebugHostBaseClass	Represents a base class.
IDebugHostPublic	Represents a symbol within the publics table of a PDB. This does not have type information associated with it. It is a name and address.
IDebugHostModuleSignature	Represents a module signature -- a definition which will match a set of modules by name and/or version
IDebugHostTypeSignature	Represents a type signature -- a definition which will match a set of types by module and/or name

The Core Host Interface: IDebugHost

The IDebugHost interface is the core interface of any data model host. It is defined as follows:

```
DECLARE_INTERFACE_(IDebugHost, IUnknown)
{
    STDMETHOD(GetHostDefinedInterface)(_COM_Outptr_ IUnknown** hostUnk) PURE;
    STDMETHOD(GetCurrentContext)(_COM_Outptr_ IDebugHostContext** context) PURE;
    STDMETHOD(GetDefaultMetadata)(_COM_Outptr_ IKeyStore** defaultMetadataStore) PURE;
}
```

GetHostDefinedInterface

The GetHostDefinedInterface method returns the host's main private interface, if such exists for the given host. For Debugging Tools for Windows, the interface returned here is an IDebugClient (cast to IUnknown).

GetCurrentContext

The GetCurrentContext method returns an interface which represents the current state of the debugger host. The exact meaning of this is left up to the host, but it typically includes things such as the session, process, and address space that is active in the user interface of the debug host. The returned context object is largely opaque to the caller but it is an important object to pass between calls to the debug host. When a caller is, for instance, reading memory, it is important to know which process and address space that memory is being read from. That notion is encapsulated in the notion of the context object which is returned from this method.

GetDefaultMetadata

The GetDefaultMetadata method returns a default metadata store that may be used for certain operations (e.g.: string conversion) when no explicit metadata has been passed. This allows the debug host to have some control over the way some data is presented. For example, the default metadata may include a PreferredRadix key, allowing the host to indicate whether ordinals should be displayed in decimal or hexadecimal if not otherwise specified.

Note that property values on the default metadata store must be manually resolved and must pass the object for which the default metadata is being queried. The GetKey method should be used in lieu of GetKeyValue.

The Status Interface: IDebugHostStatus

The IDebugHostStatus interface allows a client of the data model or the debug host to inquire about certain aspects of the debug host's status. The interface is defined as follows:

```
DECLARE_INTERFACE_(IDebugHostStatus, IUnknown)
{
    STDMETHOD(PollUserInterrupt)(_Out_ bool* interruptRequested) PURE;
}
```

PollUserInterrupt

The PollUserInterrupt method is used to inquire whether the user of the debug host has requested an interruption of the current operation. A property accessor in the data model may, for instance, call into arbitrary code (e.g.: a JavaScript method). That code may take an arbitrary amount of time. In order to keep the debug host responsive, any such code which may take an arbitrary amount of time should check for an interrupt request via calling this method. If the interruptRequested value comes back as true, the caller should immediately abort and return a result of E_ABORT.

The Context Interface: IDebugHostContext

Context is one of the most important aspects of the data model and the underlying debug host. When you hold an object, it is important to be able to know where an object came from -- what process is it in, what address space is it associated with. Knowing this information allows the correct interpretation of things like pointer values. An object of the type IDebugHostContext must be passed to many methods on the debug host. This interface can be acquired in a number of ways:

- By getting the current context of the debugger: calling the GetCurrentContext method of IDebugHost
- By getting the context of an object: calling the GetContext method of IModelObject
- By getting the context of a symbol: calling the GetContext method of IDebugHostSymbol

In addition, there are two values which have special meaning in the context of an IDebugHostContext interface which is either returned from or passed to a data model or debug host method:

nullptr: an indication that there is no context. It is perfectly valid for some objects to have no context. The Debugger object in the root namespace of the data model does not refer to anything within a specific process or address space. It has no context.

USE_CURRENT_HOST_CONTEXT: a sentinel value indicating that one should use the current UI context of the debug host. This value will never be returned from the debug host. It may, however, be passed to any debug host method which takes an input IDebugHostContext in lieu of explicitly calling the GetCurrentContext method of IDebugHost. Note that explicitly passing USE_CURRENT_HOST_CONTEXT is often more performant than explicitly getting the current context.

The contexts of a host context are largely opaque to the caller. The only operation that a caller outside the core debug host can do with a host context is to compare it to another host context.

The IDebugHostContext interface is defined as follows:

```
DECLARE_INTERFACE_(IDebugHostContext, IUnknown)
{
    STDMETHOD(IsEqualTo)(_In_ IDebugHostContext *pContext, _Out_ bool *pIsEqual) PURE;
}
```

IsEqualTo

The IsEqualTo method compares a host context to another host context. If the two contexts are equivalent, an

indication of this is returned. Note that this comparison is not interface equivalence. This compares the underlying opaque contents of the context itself.

The Error Sink: IDebugHostErrorSink

The IDebugHostErrorSink is a means by which a client can receive notifications of errors which occur during certain operations and route those errors where needed. The interface is defined as follows:

```
enum ErrorClass
{
    ErrorClassWarning,
    ErrorClassError
}
```

```
DECLARE_INTERFACE_(IDebugHostErrorSink, IUnknown)
{
    STDMETHOD(ReportError)(_In_ ErrorClass errClass, _In_ HRESULT hrError, _In_ PCWSTR message) PURE;
}
```

ReportError

The ReportError method is a callback on the error sink to notify it that an error has occurred and allow the sink to route the error to whatever UI or mechanism is appropriate.

The Host Evaluator: IDebugHostEvaluator / IDebugHostEvaluator2

One of the most important pieces of functionality which the debug host provides to clients is access to its language based expression evaluator. The IDebugHostEvaluator and IDebugHostEvaluator2 interfaces are the means to access that functionality from the debug host.

The interfaces are defined as follows:

```
DECLARE_INTERFACE_(IDebugHostEvaluator2, IDebugHostEvaluator)
{
    //
    // IDebugHostEvaluator:
    //
    STDMETHOD(EvaluateExpression)(_In_ IDebugHostContext* context, _In_ PCWSTR expression, _In_opt_
IModelObject* bindingContext, _COM_Errorptr_ IModelObject** result, _COM_Outptr_opt_result_maybenull_
IKeyStore** metadata) PURE;
    STDMETHOD(EvaluateExtendedExpression)(_In_ IDebugHostContext* context, _In_ PCWSTR expression, _In_opt_
IModelObject* bindingContext, _COM_Errorptr_ IModelObject** result, _COM_Outptr_opt_result_maybenull_
IKeyStore** metadata) PURE;
    //
    // IDebugHostEvaluator2:
    //
    STDMETHOD(AssignTo)(_In_ IModelObject* assignmentReference, _In_ IModelObject* assignmentValue,
_COM_Errorptr_ IModelObject** assignmentResult, _COM_Outptr_opt_result_maybenull_ IKeyStore**
assignmentMetadata) PURE;
}
```

EvaluateExpression

The EvaluateExpression method allows requests the debug host to evaluate a language (e.g.: C++) expression and return the resulting value of that expression evaluation boxed as an IModelObject. This particular variant of the method only allows language constructs. Any additional functionality which is presented within the expression evaluator of the debug host that is not present in the language (e.g.: LINQ query methods) is turned off for the evaluation.

EvaluateExtendedExpression

The EvaluateExtendedExpression method is similar to the EvaluateExpression method except that it turns back on additional non-language functionality which a particular debug host chooses to add to its expression evaluator. For Debugging Tools for Windows, for example, this enables anonymous types, LINQ queries, module qualifiers, format specifiers, and other non-C/C++ functionality.

IDebugHostEvaluator2

AssignTo

The AssignTo method performs assignment according to the semantics of the language being debugged.

The Host Extensibility Interface: IDebugHostExtensibility

Certain functionality of the debug host is optionally subject to extensibility. This may, for instance, include the expression evaluator. The IDebugHostExtensibility interface is the means by which these extensibility points are accessed. The interface is defined as follows:

```
DECLARE_INTERFACE_(IDebugHostExtensibility, IUnknown)
{
    STDMETHOD(CreateFunctionAlias)(_In_ PCWSTR aliasName, _In_ IModelObject *functionObject) PURE;
    STDMETHOD(DestroyFunctionAlias)(_In_ PCWSTR aliasName) PURE;
}
```

CreateFunctionAlias

The CreateFunctionAlias method creates a "function alias", a "quick alias" for a method implemented in some extension. The meaning of this alias is host specific. It may extend the host's expression evaluator with the function or it may do something entirely different.

DestroyFunctionAlias

The DestroyFunctionAlias method undoes a prior call to the CreateFunctionAlias method. The function will no longer be available under the quick alias name.

Accessing the Data Model

First and foremost, the data model extensibility APIs are designed to be neutral to the application (typically a debugger) which acts as a host of the data model. In theory, any application can host the data model by providing a set of host APIs that expose the type system of the application's debug target(s) and a set of projected objects into the namespace of the data model about what targets, processes, threads, etc... are in those debug target(s).

While the data model APIs -- those that begin IDataModel, *IDebugHost*, and the offshoots of IModelObject -- are designed to be portable, they do not define what a "debugger extension" is. Today, a component that wishes to extend Debugging Tools for Windows and the engine it provides must write an engine extension in order to get access to the data model. That engine extension needs only to be an engine extension in so much as that is the loading and bootstrapping mechanism for the extension. As such, a minimal implementation would provide:

- **DebugExtensionInitialize**: A method which utilizes a created IDebugClient to get access to the data model and sets up object model manipulations.
- **DebugExtensionUninitialize**: A method which undoes the object model manipulations which were performed in DebugExtensionInitialize.
- **DebugExtensionCanUnload**: A method which returns whether the extension can unload. If there are still live COM objects in the extension, it must indicate this. This is the debugger's equivalent of COM's DllCanUnloadNow. If this returns the S_FALSE indication of inability to unload, the debugger can query this later to see if an unload is safe or it may reinitialize the extension by calling DebugExtensionInitialize again. The extension must be prepared to handle both paths.
- **DebugExtensionUnload**: A method which does any final cleanup required right before the DLL unloads

The Bridge Interface: IHostDataModelAccess

As mentioned, when DebugExtensionInitialize is called, it creates a debug client and gets access to the data model. Such access is provided by a bridge interface between the legacy IDebug* interfaces of Debugging Tools for Windows and the data model. This bridge interface is 'IHostDataModelAccess and is defined as follows:

```
DECLARE_INTERFACE_(IHostDataModelAccess, IUnknown)
{
   STDMETHOD(GetDataModel)(_COM_Outptr_ IDataModelManager** manager, _COM_Outptr_ IDebugHost** host) PURE;
}
```

GetDataModel

The GetDataModel method is the method on the bridge interface which provides access to both sides of the data model: The debug host (the lower edge of the debugger) is expressed by the returned IDebugHost interface The data model's main component -- the data model manager is expressed by the returned IDataModelManager interface

Debugger Data Model System Interfaces

The Data Model Host

The Debugger Data Model is designed to be a componentized system which can be hosted in a variety of different contexts. Normally, the Data Model is hosted in the context of a debugger application. In order to be a host of the data model, a number of interfaces need to be implemented to expose core aspects of the debugger: its targeting, its memory spaces, its evaluator, its symbolic and type system, etc... While these interfaces are implemented by any application wishing to host the data model, they are consumed by both the core data model as well as any extension which interoperates with the data model.

The type system and symbolic interfaces are:

INTERFACE NAME	DESCRIPTION
IDebugHostSymbols	Core interface which provides access to and resolution of symbols
IDebugHostSymbol / IDebugHostSymbol2	Represents a single symbol of any kind. The particular symbol is a derivation of this interface.
IDebugHostModule	Represents a module loaded within a process. This is a kind of symbol.
IDebugHostType / IDebugHostType2	Represents a native/language type.
IDebugHostConstant	Represents a constant within symbolic information (e.g.: a non-type template argument in C++)
IDebugHostField	Represents a field within a structure or class.
IDebugHostData	Represents data within a module (were this within a structure or class it would be an IDebugHostField)
IDebugHostBaseClass	Represents a base class.

INTERFACE NAME	DESCRIPTION
IDebugHostPublic	Represents a symbol within the publics table of a PDB. This does not have type information associated with it. It is a name and address.
IDebugHostModuleSignature	Represents a module signature -- a definition which will match a set of modules by name and/or version
IDebugHostTypeSignature	Represents a type signature -- a definition which will match a set of types by module and/or name

The other core interfaces are:

INTERFACE NAME	DESCRIPTION
IDebugHost	The core interface to the debug host.
IDebugHostStatus	An interface allowing a client to query for the status of the host.
IDebugHostContext	An abstraction of a context within the host (e.g.: a particular target, a particular process, a particular address space, etc...)
IDebugHostErrorSink	An interface implemented by callers to receive errors from certain portions of the host and data model
IDebugHostEvaluator / IDebugHostEvaluator2	The debug host's expression evaluator.
IDebugHostExtensibility	An interface for extending the capabilities of the host or portions of it (such as the expression evaluator).

The Main Symbolic Interface: IDebugHostSymbols

The IDebugHostSymbols interface is the main starting point to access symbols in the debug target. This interface can be queried from an instance of IDebugHost and is defined as follows:

```
DECLARE_INTERFACE_(IDebugHostSymbols, IUnknown)
{
    STDMETHOD(CreateModuleSignature)(_In_z_ PCWSTR pwszModuleName, _In_opt_z_ PCWSTR pwszMinVersion,
_In_opt_z_ PCWSTR pwszMaxVersion, _Out_ IDebugHostModuleSignature** ppModuleSignature) PURE;
    STDMETHOD(CreateTypeSignature)(_In_z_ PCWSTR signatureSpecification, _In_opt_ IDebugHostModule* module,
_Out_ IDebugHostTypeSignature** typeSignature) PURE;
    STDMETHOD(CreateTypeSignatureForModuleRange)(_In_z_ PCWSTR signatureSpecification, _In_z_ PCWSTR
moduleName, _In_opt_z_ PCWSTR minVersion, _In_opt_z_ PCWSTR maxVersion, _Out_ IDebugHostTypeSignature**
typeSignature) PURE;
    STDMETHOD(EnumerateModules)(_In_ IDebugHostContext* context, _COM_Outptr_ IDebugHostSymbolEnumerator**
moduleEnum) PURE;
    STDMETHOD(FindModuleByName)(_In_ IDebugHostContext* context, _In_z_ PCWSTR moduleName, _COM_Outptr_
IDebugHostModule **module) PURE;
    STDMETHOD(FindModuleByLocation)(_In_ IDebugHostContext* context, _In_ Location moduleLocation,
_COM_Outptr_ IDebugHostModule **module) PURE;
    STDMETHOD(GetMostDerivedObject)(_In_opt_ IDebugHostContext *pContext, _In_ Location location, _In_
IDebugHostType* objectType, _Out_ Location* derivedLocation, _Out_ IDebugHostType** derivedType) PURE;
}
```

CreateModuleSignature

The CreateModuleSignature method creates a signature which can be used to match a set of specific modules by name and optionally, by version. There are three components to a module signature:

- A name: a matching module must have a name which is an exact case insensitive match against the name in the signature
- A minimum version: if specified, a matching module must have a minimum version which is at least as high as this version. Versions are specified in "A.B.C.D" format with each subsequent portion being less important than the prior. Only the first segment is mandatory.
- A maximum version: if specified, a matching module must have a maximum version which is no higher than this version. Versions are specified in "A.B.C.D" format with each subsequent portion being less important than the prior. Only the first segment is mandatory.

CreateTypeSignature

The CreateTypeSignature method creates a signature which can be used to match a set of concrete types by containing module and type name. The format of the type name signature string is specific to the language being debugged (and debug host). For C/C++, the signature string is equivalent to a NatVis Type Specification. That is, the signature string is a type name where wildcards (specified as *) are allowed for template arguments.

CreateTypeSignatureForModuleRange

The CreateTypeSignatureForModuleRange method creates a signature which can be used to match a set of concrete types by module signature and type name. This is similar to the CreateTypeSignature method excepting that instead of passing a specific module to match for the signature, the caller passes the arguments necessary to create a module signature (as if the module signature were created with the CreateModuleSignature method).

EnumerateModules

The EnumerateModules method creates an enumerator which will enumerate every module available in a particular host context. That host context might encapsulate a process context or it might encapsulate something like the Windows kernel.

FindModuleByName

The FindModuleByName method will look through the given host context and locate a module which has the specified name and return an interface to it. It is legal to search for the module by name with or without the file extension.

FindModuleByLocation

The FindModuleByLocation method will look through the given host context and determine what module contains the address given by the specified location. It will then return an interface to such module.

GetMostDerivedObject

The GetMostDerivedObject will use the type system of the debugger to determine the runtime type of an object from its static type. This method will only use symbolic information and heuristics available at the type system layer in order to perform this analysis. Such information may include C++ RTTI (run time type information) or analysis of the shape of the virtual function tables of the object. It does not include things such as the preferred runtime type concept on an IModelObject. If the analysis cannot find a runtime type or cannot find a runtime type different from the static type passed into the method, the input location and type may be passed out. The method will not fail for these reasons.

The Core Individual Symbol Interface: IDebugHostSymbol

Every symbol that can be returned from the data model host will derive in some fashion from IDebugHostSymbol. This is the core interface that every symbol implements regardless of the kind of symbol. Depending on the kind of symbol, a given symbol may implement a set of other interfaces which return attributes

more unique to the particular kind of symbol represented by this interface. The IDebugHostSymbol2 / IDebugHostSymbol interface is defined as follows:

```
DECLARE_INTERFACE_(IDebugHostSymbol2, IDebugHostSymbol)
{
    //
    // IDebugHostSymbol:
    //
    STDMETHOD(GetContext)(_COM_Outptr_ IDebugHostContext** context) PURE;
    STDMETHOD(EnumerateChildren)(_In_ SymbolKind kind, _In_opt_z_ PCWSTR name, _Out_
IDebugHostSymbolEnumerator **ppEnum) PURE;
    STDMETHOD(GetSymbolKind)(_Out_ SymbolKind *kind) PURE;
    STDMETHOD(GetName)(_Out_ BSTR* symbolName) PURE;
    STDMETHOD(GetType)(_Out_ IDebugHostType** type) PURE;
    STDMETHOD(GetContainingModule)(_Out_ IDebugHostModule **containingModule) PURE;
    STDMETHOD(CompareAgainst)(_In_ IDebugHostSymbol *pComparisonSymbol, _In_ ULONG comparisonFlags, _Out_ bool
*pMatches) PURE;
    //
    // IDebugHostSymbol2
    //
    STDMETHOD(EnumerateChildrenEx)(_In_ SymbolKind kind, _In_opt_z_ PCWSTR name, _In_opt_ SymbolSearchInfo*
searchInfo, _Out_ IDebugHostSymbolEnumerator **ppEnum) PURE;
}
```

It is very important to note that this interface represents many kinds of symbols -- delineated by the SymbolKind enumeration which has values as follows:

ENUMARANT	MEANING
Symbol	Unspecified symbol type
SymbolModule	The symbol is a module and can be queried for IDebugHostModule
SymbolType	The symbol is a type and can be queried for IDebugHostType
SymbolField	The symbol is a field (a data member within a structure or class) and can be queried for IDebugHostField
SymbolConstant	The symbol is a constant value and can be queried for IDebugHostConstant
SymbolData	The symbol is data which is not a member of a structure or class and is queryable for IDebugHostData
SymbolBaseClass	The symbol is a base class and is queryable for IDebugHostBaseClass
SymbolPublic	The symbol is an entry in a module's publics table (having no type information) and is queryable for IDebugHostPublic
SymbolFunction	The symbol is a function and is queryable for IDebugHostData

GetContext

The GetContext method returns the context where the symbol is valid. While this will represent things such as the debug target and process/address space in which the symbol exists, it may not be as specific as a context retrieved from other means (e.g.: from an *IModelObject*).

EnumerateChildren

The EnumerateChildren method returns an enumerator which will enumerate all children of a given symbol. For a C++ type, for example, the base classes, fields, member functions, and the like are all considered children of the type symbol.

The Module Interface: IDebugHostModule

The debugger's notion of a module that is loaded within some address space is represented in two distinct ways in the data model: At the type system level via the IDebugHostModule interface. Here, a module is a symbol and core attributes of the module are interface method calls Projected at the data model level via the Debugger.Models.Module data model. This is an extensible encapsulation of the type system IDebugHostModule representation of a module.

The IDebugHostModule interface is defined as follows (ignoring methods that are generic to IDebugHostSymbol):

```
DECLARE_INTERFACE_(IDebugHostModule, IDebugHostSymbol)
{
    //
    // IDebugHostModule:
    //
    STDMETHOD(GetImageName)(_In_ bool allowPath, _Out_ BSTR* imageName) PURE;
    STDMETHOD(GetBaseLocation)(_Out_ Location* moduleBaseLocation) PURE;
    STDMETHOD(GetVersion)(_Out_opt_ ULONG64* fileVersion, _Out_opt_ ULONG64* productVersion) PURE;
    STDMETHOD(FindTypeByName)(_In_z_ PCWSTR typeName, _Out_ IDebugHostType** type) PURE;
    STDMETHOD(FindSymbolByRVA)(_In_ ULONG64 rva, _Out_ IDebugHostSymbol** symbol) PURE;
    STDMETHOD(FindSymbolByName)(_In_z_ PCWSTR symbolName, _Out_ IDebugHostSymbol** symbol) PURE;
}
```

GetImageName

The GetImageName method returns the image name of the module. Depending on the value of the allowPath argument, the returned image name may or may not include the full path to the image.

GetBaseLocation

The GetBaseLocation method returns the base load address of the module as a location structure. The returned location structure for a module will typically refer to a virtual address.

GetVersion

The GetVersion method returns version information about the module (assuming that such information can successfully be read out of the headers). If a given version is requested (via a non-nullptr output pointer) and it cannot be read, an appropriate error code will be returned from the method call.

FindTypeByName

The FindTypeByName method finds a type defined within the module by the type name and returns a type symbol for it. This method may return a valid IDebugHostType which would never be returned via explicit recursion of children of the module. The debug host may allow creation of derivative types -- types not ever used within the module itself but derived from types that are. As an example, if the structure MyStruct is defined in the symbols of the module but the type MyStruct ** is never used, the FindTypeByName method may legitimately return a type symbol for MyStruct ** despite that type name never explicitly appearing in the symbols for the module.

FindSymbolByRVA

The FindSymbolByRVA method will find a single matching symbol at the given relative virtual address within the

module. If there is not a single symbol at the supplied RVA (e.g.: there are multiple matches), an error will be returned by this method. Note that this method will prefer returning a private symbol over a symbol in the publics table.

FindSymbolByName

The FindSymbolByName method will find a single global symbol of the given name within the module. If there is not a single symbol matching the given name, an error will be returned by this method. Note that this method will prefer returning a private symbol over a symbol in the publics table.

Access to the Type System: IDebugHostType2 / IDebugHostType

A given language/native type is described by the IDebugHostType2 or IDebugHostType interfaces. Note that some of the methods on these interfaces only apply for specific kinds of types. A given type symbol may refer to one of the following types as described by the TypeKind enumeration:

TYPE KIND	DESCRIPTION
TypeUDT	A user defined type (a struct, class, union, etc...). A model object which has a native type whose kind is TypeUDT has a canonical representation of ObjectTargetObject where the type is always kept inside the corresponding IModelObject.
TypePointer	A pointer. A model object which has a native type whose kind is TypePointer has a canonical representation of ObjectIntrinsic where the pointer's value is zero extended to VT_UI8 and kept as intrinsic data in this 64-bit form. Any type symbol of TypePointer has a base type (as returned by the GetBaseType method) of the type that the pointer points to.
TypeMemberPointer	A pointer to class member. A model object which has a native type whose kind is TypeMemberPointer has a canonical representation which is intrinsic (the value being the same as the pointer value). The exact meaning of this value is compiler/debug host specific.
TypeArray	An array. A model object which has a native type whose kind is TypeArray has a canonical representation of ObjectTargetObject. The base address of the array is the object's location (retrieved via the GetLocation method) and the type of the array is always kept. Any type symbol of TypeArray has a base type (as returned by the GetBaseType method) of the type that the array is an array of.
TypeFunction	A function.
TypeTypedef	A typedef. A model object which has a native type whose kind would otherwise be TypeTypedef has a canonical representation identical to the canonical representation of the final type underlying the typedef. This appears completely transparent to the end user of the object and the type information unless the explicit typedef methods of IDebugHostType2 are utilized to query typedef information or there is an explicit data model registered against the typedef. Note that the GetTypeKind method will never return TypeTypedef. Every method will return what the final type underlying the typedef would return. There are typedef specific methods on IDebugHostType2 which can be used to get the typedef specific information.

TYPE KIND	DESCRIPTION
TypeEnum	An enum. A model object which has a native type whose kind is TypeEnum has a canonical representation of ObjectIntrinsic where the value and type of the intrinsic is identical to the enum value.
TypeIntrinsic	An intrinsic (base type). A model object which has a native type whose kind is TypeIntrinsic has a canonical representation of ObjectIntrinsic. The type information may or may not be kept -- particularly if the underlying type is fully described by the variant data type (VT_*) of the intrinsic data stored in the IModelObject

The overall IDebugHostType2 / IDebugHostType interface is defined as follows (excluding IDebugHostSymbol methods):

```
DECLARE_INTERFACE_(IDebugHostType2, IDebugHostType)
{
    //
    // IDebugHostType:
    //
    STDMETHOD(GetTypeKind)(_Out_ TypeKind *kind) PURE;
    STDMETHOD(GetSize)(_Out_ ULONG64* size) PURE;
    STDMETHOD(GetBaseType)(_Out_ IDebugHostType** baseType) PURE;
    STDMETHOD(GetHashCode)(_Out_ ULONG* hashCode) PURE;
    STDMETHOD(GetIntrinsicType)(_Out_opt_ IntrinsicKind *intrinsicKind, _Out_opt_ VARTYPE *carrierType) PURE;
    STDMETHOD(GetBitField)(_Out_ ULONG* lsbOfField, _Out_ ULONG* lengthOfField) PURE;
    STDMETHOD(GetPointerKind)(_Out_ PointerKind* pointerKind) PURE;
    STDMETHOD(GetMemberType)(_Out_ IDebugHostType** memberType) PURE;
    STDMETHOD(CreatePointerTo)(_In_ PointerKind kind, _COM_Outptr_ IDebugHostType** newType) PURE;
    STDMETHOD(GetArrayDimensionality)(_Out_ ULONG64* arrayDimensionality) PURE;
    STDMETHOD(GetArrayDimensions)(_In_ ULONG64 dimensions, _Out_writes_(dimensions) ArrayDimension
*pDimensions) PURE;
    STDMETHOD(CreateArrayOf)(_In_ ULONG64 dimensions, _In_reads_(dimensions) ArrayDimension *pDimensions,
_COM_Outptr_ IDebugHostType** newType) PURE;
    STDMETHOD(GetFunctionCallingConvention)(_Out_ CallingConventionKind* conventionKind) PURE;
    STDMETHOD(GetFunctionReturnType)(_COM_Outptr_ IDebugHostType** returnType) PURE;
    STDMETHOD(GetFunctionParameterTypeCount)(_Out_ ULONG64* count) PURE;
    STDMETHOD(GetFunctionParameterTypeAt)(_In_ ULONG64 i, _Out_ IDebugHostType** parameterType) PURE;
    STDMETHOD(IsGeneric)(_Out_ bool* isGeneric) PURE;
    STDMETHOD(GetGenericArgumentCount)(_Out_ ULONG64* argCount) PURE;
    STDMETHOD(GetGenericArgumentAt)(_In_ ULONG64 i, _Out_ IDebugHostSymbol** argument) PURE;
    //
    // IDebugHostType2:
    //
    STDMETHOD(IsTypedef)(_Out_ bool* isTypedef) PURE;
    STDMETHOD(GetTypedefBaseType)(_Out_ IDebugHostType2** baseType) PURE;
    STDMETHOD(GetTypedefFinalBaseType)(_Out_ IDebugHostType2** finalBaseType) PURE;
    STDMETHOD(GetFunctionVarArgsKind)(_Out_ VarArgsKind* varArgsKind) PURE;
}
```

IDebugHostType2/IDebugHostType General Methods

The following IDebugHostType methods are general to any type regardless of what kind is returned from the GetTypeKind method:

```
STDMETHOD(GetTypeKind)(_Out_ TypeKind *kind) PURE;
STDMETHOD(GetSize)(_Out_ ULONG64* size) PURE;
STDMETHOD(GetBaseType)(_Out_ IDebugHostType** baseType) PURE;
STDMETHOD(GetHashCode)(_Out_ ULONG* hashCode) PURE;
```

GetTypeKind

The GetTypeKind method returns what kind of type (pointer, array, intrinsic, etc...) the symbol refers to.

GetSize

The GetSize method returns the size of the type (as if one had done sizeof(type) in C++).

GetBaseType

If the type is a derivative of another single type (e.g.: as MyStruct * is derived from MyStruct'), the GetBaseType method returns the base type of the derivation. For pointers, this returns the type pointed to. For arrays, this returns what the array is an array of. If the type is not such a derivative type, an error is returned.

GetHashCode

The GetHashCode method returns a 32-bit hash code for the type. With the exception of a global match (e.g.: a type signature equivalent to * which matches everything if permitted by the host), any type instance which can match a particular type signature must return the same hash code. This method is used in conjunction with type signatures in order to match type signatures to type instances.

IDebugHostType2/IDebugHostType Intrinsic Methods

The following IDebugHostType methods are specific to intrinsic types (or types which hold intrinsic data such as enums):

```
STDMETHOD(GetIntrinsicType)(_Out_opt_ IntrinsicKind *intrinsicKind, _Out_opt_ VARTYPE *carrierType) PURE;
```

GetIntrinsicType

The GetIntrinsicType method returns information about what kind of intrinsic the type is. Two values are returned out of this method:

- The intrinsic kind indicates the overall type (e.g.: integer, unsigned, floating point) but not the size of the type (e.g.: 8 bit, 16 bit, 32 bit, 64 bit)
- The carrier type indicates how the intrinsic kind packs into a VARIANT structure. This is a VT_* constant.

The combination of the two values provides the full set of information about the intrinsic.

IDebugHostType2/IDebugHostType Bitfield Methods

The following IDebugHostType methods are specific to types which store data in bitfields. Information about bitfield placement within an intrinsic is stored as part of the type symbol in the data model rather than being an attribute of the location.

```
STDMETHOD(GetBitField)(_Out_ ULONG* lsbOfField, _Out_ ULONG* lengthOfField) PURE;
```

GetBitField

If a given member of a data structure is a bitfield (e.g.: ULONG MyBits:8), the type information for the field carries with it information about the bitfield placement. The GetBitField method can be used to retrieve that information. This method will fail on any type which is not a bitfield. This is the only reason the method will fail. Simply calling this method and looking at success/failure is sufficient to distinguish a bit field from a non-bit field. If a given type does happen to be a bitfield, the field positions are defined by the half open set *(lsbOfField + lengthOfField : lsbOfField]*

IDebugHostType2/IDebugHostType Pointer Related Methods

The following IDebugHostType methods are specific to pointer types. Such are types where GetTypeKind returns TypePointer or TypeMemberPointer':

```
STDMETHOD(GetPointerKind)(_Out_ PointerKind* pointerKind) PURE;
STDMETHOD(GetMemberType)(_Out_ IDebugHostType** memberType) PURE;
```

GetPointerKind

For types which are pointers, the GetPointerKind method returns the kind of pointer. This is defined by the PointerKind enumeration.

GetMemberType

For types which are pointer-to-member (as indicated by a type kind of TypeMemberPointer), the GetMemberType method returns the class the pointer is a pointer-to-member of.

IDebugHostType2/IDebugHostType Array Related Methods

Arrays are types where GetTypeKind returns TypeArray. Note that arrays as defined by the debug host's type system are not the same as the single dimensional, zero index based, packed linear one dimensional arrays that C utilizes. C style arrays fit into the definition but the overall scope of an array is broader in IDebugHostType. An array in the debug host can be multi-dimensional and each dimension within the array is defined by a descriptor known as an ArrayDimensionThis descriptor has the following fields:

FIELD	MEANING
LowerBound	The base index of the array as a signed 64-bit value. For a C style array, this will always be zero. It need not be. An individual dimension of an array can be considered to start at any 64-bit index, even a negative one.
Length	The length of the array dimension as an unsigned 64-bit value. The indicies of the array span the half open set [LowerBound, LowerBound + Length).
Stride	Defines the stride of the array dimension. For an increase of one (from N to N + 1) in the index of this dimension, this indicates how many bytes to move forward in memory. For a C style array, this would be the size of each element of the array. It does not need to be. Padding between elements can be expressed as a stride greater than the size of each individual element. For multi-dimensional arrays, this value would indicate how to move an entire dimension forward. Consider an M x N matrix. This might be described in row-major form as two dimensions:

```
{ [LowerBound: 0, Length: M, Stride: N \* sizeof(element)], [LowerBound: 0, Length: N, Stride:
sizeof(element)]}
```

or it might be alternatively be described in column-major form as two dimensions:

```
{ [LowerBound: 0, Length: M, Stride: sizeof(element)], [LowerBound: 0, Length: N, Stride: M \*
sizeof(element)]}
```

The ArrayDimension concept allows this degree of flexibility.

The following IDebugHostType methods are specific to array types.

```
STDMETHOD(GetArrayDimensionality)(\_Out_ ULONG64\* arrayDimensionality) PURE;
STDMETHOD(GetArrayDimensions)(\_In_ ULONG64 dimensions, \_Out_writes_(dimensions) ArrayDimension
\*pDimensions) PURE;
```

GetArrayDimensionality

The GetArrayDimensionality method returns the number of dimensions that the array is indexed in. For C style arrays, the value returned here will always be 1.

GetArrayDimensions

The GetArrayDimensions method returns a set of descriptors, one for each dimension of the array as indicated by the GetArrayDimensionality method. Each descriptor is an ArrayDimension structure which describes the starting index, length, and forward stride of each array dimension. This allows descriptions of significantly more powerful array constructs than are allowed in the C type system.

For C-style arrays, a single array dimension is returned here with values which are always:

- LowerBound = 0
- Length = ARRAYSIZE(array)
- Stride = sizeof(elementType)

IDebugHostType2/IDebugHostType Function Related Methods

Types which indicate that they are function types via a kind of TypeFunction support the following methods in both IDebugHostType and IDebugHostType2.

```
//
// IDebugHostType:
//
STDMETHOD(GetFunctionCallingConvention)(_Out_ CallingConventionKind* conventionKind) PURE;
STDMETHOD(GetFunctionReturnType)(_COM_Outptr_ IDebugHostType** returnType) PURE;
STDMETHOD(GetFunctionParameterTypeCount)(_Out_ ULONG64* count) PURE;
STDMETHOD(GetFunctionParameterTypeAt)(_In_ ULONG64 i, _Out_ IDebugHostType** parameterType) PURE;
//
// IDebugHostType2:
//
STDMETHOD(GetFunctionVarArgsKind)(_Out_ VarArgsKind* varArgsKind) PURE;
```

GetFunctionCallingConvention

The GetFunctionCallingConvention method returns the calling convention of the function. Such is returned as a member of the CallingConventionKind enumeration.

GetFunctionReturnType

The GetFunctionReturnType method returns the return type of the function.

GetFunctionParameterTypeCount

The GetFunctionParameterTypeCount method returns the number of arguments that the function takes. Note that the C/C++ ellipsis based variable argument marker is not considered in this count. The presence of such must be detected via the GetFunctionVarArgsKind method. This will only include arguments before the ellipsis.

GetFunctionParameterTypeAt

The GetFunctionParameterTypeAt method returns the type of the i-th argument to the function.

The GetFunctionVarArgsKind method returns whether a given function utilizes a variable argument list, and if so, what style of variable arguments it utilizes. Such is defined by a member of the VarArgsKind enumeration defined

as follows:

ENUMERANT	MEANING
VarArgsNone	The function does not take any variable arguments.
VarArgsCStyle	The function is a C-style varargs function (returnType(arg1, arg2, ...)). The number of arguments reported by the function does not include the ellipsis argument. Any variable argument passing occurs after the number of arguments returned by the GetFunctionParameterTypeCount method.

IDebugHostType2 GetFunctionVarArgsKind

The GetFunctionVarArgsKind method returns whether a given function utilizes a variable argument list, and if so, what style of variable arguments it utilizes. Such is defined by a member of the VarArgsKind enumeration defined as follows:

IDebugHostType2/IDebugHostType Typedef Related Methods

Any type which is a typedef will behave as if the type is the final type underlying the typedef. This means that methods such as GetTypeKind will not indicate that the type is a typedef. Likewise, GetBaseType will not return the type the definition refers to. They will instead indicate behave as if they were called on the final definition underlying the typedef. As an example:

```
typedef MYSTRUCT *PMYSTRUCT;
typedef PMYSTRUCT PTRMYSTRUCT;
```

An IDebugHostType for 'either PMYSTRUCT or PTRMYSTRUCT will report the following information:

- The GetTypeKind method will return TypePointer. The final underlying type MYSTRUCT * is indeed a pointer.
- The 'GetBaseType method will return a type for MYSTRUCT. The underlying type of MYSTRUCT * is MYSTRUCT.

The only difference here is how the typedef specific methods on IDebugHostType2 behave. Those methods are:

```
STDMETHOD(IsTypedef)(_Out_ bool* isTypedef) PURE;
STDMETHOD(GetTypedefBaseType)(_Out_ IDebugHostType2** baseType) PURE;
STDMETHOD(GetTypedefFinalBaseType)(_Out_ IDebugHostType2** finalBaseType) PURE;
```

In this example:

- The IsTypedef method will return true for both PMYSTRUCT and PTRMYSTRUCT
- The GetTypedefBaseType method will return MYSTRUCT * for PMYSTRUCT and PMYSTRUCT for PTRMYSTRUCT
- The GetTypedefFinalBaseType method will return MYSTRUCT * for both types

IsTypedef

The IsTypedef method is the only method capable of seeing whether a type is a typedef. The GetTypeKind method will behave as if called on the underlying type.

GetTypedefBaseType

The GetTypedefBaseType method will return what the immediate definition of the typedef. In the examples described in the documentation:

```
typedef MYSTRUCT *PMYSTRUCT;
typedef PMYSTRUCT PTRMYSTRUCT;
```

this method will return MYSTRUCT * for PMYSTRUCT and PMYSTRUCT for PTRMYSTRUCT.

GetTypedefFinalBaseType

The GetTypedefFinalBaseType method will return the final type that the typedef is a definition for. If the typedef is a definition of another typedef, this will continue to follow the definition chain until it reaches a type which is not a typedef and that type will be returned. In the examples described in the documentation:

```
typedef MYSTRUCT *PMYSTRUCT;
typedef PMYSTRUCT PTRMYSTRUCT;
```

this method will return MYSTRUCT * when called on either PMYSTRUCT or PTRMYSTRUCT.

IDebugHostType2/IDebugHostType Type Creation Methods

```
STDMETHOD(CreatePointerTo)(_In_ PointerKind kind, _COM_Outptr_ IDebugHostType** newType) PURE;
STDMETHOD(CreateArrayOf)(_In_ ULONG64 dimensions, _In_reads_(dimensions) ArrayDimension *pDimensions,
_COM_Outptr_ IDebugHostType** newType) PURE;
```

Constant Symbol Values: IDebugHostConstant

For locations where constant values are present in symbolic information (where a particular value is a symbol which may or may not be a constant value), the IDebugHostConstant interface expresses the notion of such a constant. This is typically used in places like template arguments where a given argument is typically a type but may instead be a non-type template argument (e.g.: a constant).

The IDebugHostConstant interface is defined as follows (ignoring generic methods implemented by IDebugHostSymbol):

```
DECLARE_INTERFACE_(IDebugHostConstant, IDebugHostSymbol)
{
    STDMETHOD(GetValue)(_Out_ VARIANT* value) PURE;
}
```

GetValue

The GetValue method returns the value of the constant packed into a VARIANT. It is important to note that the GetType method on IDebugHostSymbol may return a specific type symbol for the constant. In such cases, there is no guarantee that the packing of the constant value as defined by the type symbol is the same as the packing as returned by the GetValue method here.

Data Member Access: IDebugHostField

The IDebugHostField class represents a symbol which is a data member of a class, structure, union, or other type construct. It does not represent free data (e.g.: global data). The interface is defined as follows (ignoring methods generic to IDebugHostSymbol):

```
DECLARE_INTERFACE_(IDebugHostField, IDebugHostSymbol)
{
    STDMETHOD(GetLocationKind)(_Out_ LocationKind *locationKind) PURE;
    STDMETHOD(GetOffset)(_Out_ ULONG64* offset) PURE;
    STDMETHOD(GetLocation)(_Out_ Location* location) PURE;
    STDMETHOD(GetValue)(_Out_ VARIANT* value) PURE;
}
```

GetLocationKind

The GetLocationKind method returns what kind of location the symbol is at according to the LocationKind enumeration. Such enumeration can be one of the following values:

ENUMERANT	MEANING
LocationMember	The field is a regular data member of a class, structure, union, or other type construct. It has an offset which is relative to base address of the containing type construct. Such base address is typically represented by the this pointer. The offset of the field can be retrieved via the GetOffset method. The GetLocation and GetValue methods will fail for a field which is LocationMember.
LocationStatic	The field is static and has its own address. The GetLocation method will return the abstract location (e.g.: address) of the static field. The GetOffset and GetValue methods will fail for a field which is LocationStatic.
LocationConstant	The field is a constant and has a value. The GetValue method will return the value of the constant. The GetOffset and GetLocation methods will fail for a field which is LocationConstant
LocationNone	The field has no location. It may have been optimized out by the compiler or it may be a static field which is declared but never defined. Regardless of how such a field came to be, it has no physical presence or value. It is only in the symbols. All acquisition methods (GetOffset, GetLocation, and GetValue) will fail for a field which is LocationNone.

GetOffset

For fields which have an offset (e.g.: fields whose location kind indicates LocationMember), the GetOffset method will return the offset from the base address of the containing type (the this pointer) to the data for the field itself. Such offsets are always expressed as unsigned 64-bit values. If the given field does not have a location which is an offset from the base address of the containing type, the GetOffset method will fail.

GetLocation

For fields which have an address regardless of the particular type instance (e.g.: fields whose location kind indicates LocationStatic), the GetLocation method will return the abstract location (address) of the field. If the given field does not have a static location, the GetLocation method will fail.

GetValue

For fields which have a constant value defined within the symbolic information (e.g.: fields whose location kind indicates LocationConstant), the GetValue method will return the constant value of the field. If the given field does not have a constant value, the GetValue method will fail.

Free Data Access: IDebugHostData

Data in modules which is not a member of another type is represented by the IDebugHostData interface. That interface is defined as follows (ignoring methods generic to IDebugHostSymbol):

```
DECLARE_INTERFACE_(IDebugHostData, IDebugHostSymbol)
{
    STDMETHOD(GetLocationKind)(_Out_ LocationKind *locationKind) PURE;
    STDMETHOD(GetLocation)(_Out_ Location* location) PURE;
    STDMETHOD(GetValue)(_Out_ VARIANT* value) PURE;
}
```

All of these methods are semantically equivalent to their counterparts in IDebugHostField. The only difference is that the GetLocationKind method will never return LocationMember for free data.

GetLocationKind

The GetLocationKind method returns what kind of location the symbol is at according to the LocationKind enumeration. The description of this enumeration can be found in the documentation for IDebugHostField.

GetLocation

For data which has an address, the GetLocation method will return the abstract location (address) of the field. If the given data does not have a static location, the GetLocation method will fail.

GetValue

For datawhich has a constant value defined within the symbolic information (e.g.: data whose location kind indicates LocationConstant), the GetValue method will return the constant value of the field. If the given data does not have a constant value, the GetValue method will fail.

Base Classes: IDebugHostBaseClass

The inheritance hierarchy of a given type is expressed through children of a type symbol. If a given type derives (inheritance wise) from one or more types, there will be one or more SymbolBaseClass children of the type symbol for the type. Each of those SymbolBaseClass symbols represent immediate inheritance from a particular type. The name of the base class is both the name of the SymbolBaseClass symbol as well as that of the type symbol for the base class. The GetType method on the SymbolBaseClass symbol can be used to get the type symbol for the base class itself. The full inheritance hierarchy can be traversed by recursively exploring SymbolBaseClass child symbols. Each of these base class symbols is expressed by the IDebugHostBaseClass interface which is defined as follows (ignoring methods generic to IDebugHostSymbol):

```
DECLARE_INTERFACE_(IDebugHostBaseClass, IDebugHostSymbol)
{
    STDMETHOD(GetOffset)(_Out_ ULONG64* offset) PURE;
}
```

GetOffset

The GetOffset method returns the offset of the base class from the base address of the derived class. Such offset may be zero or may be a positive unsigned 64-bit value.

Public Symbols: IDebugHostPublic

Public symbols represent things in the public table within a symbol file. They are, in effect, export addresses. There is no type information associated with a public symbol -- only an address. Unless a public symbol is explicitly requested by the caller, the debug host prefers to return private symbols for every inquiry. A public symbol is expressed by the IDebugHostPublic interface which is defined as follows (ignoring methods which are generic to

IDebugHostSymbol):

```
DECLARE_INTERFACE_(IDebugHostPublic, IDebugHostSymbol)
{
    STDMETHOD(GetLocationKind)(_Out_ LocationKind *locationKind) PURE;
    STDMETHOD(GetLocation)(_Out_ Location* location) PURE;
}
```

All of these methods are semantically equivalent to their counterparts in IDebugHostField. The only difference is that the GetLocationKind method will never return LocationMember or LocationConstant for such symbols.

GetLocationKind

The GetLocationKind method returns what kind of location the symbol is at according to the LocationKind enumeration. The description of this enumeration can be found in the documentation for IDebugHostField.

GetLocation

For data which has an address, the GetLocation method will return the abstract location (address) of the field. If the given public does not have a static location, the GetLocation method will fail.

Module Signatures and Version Matching: IDebugHostModuleSignature

Module signatures represent a means to check whether a given module meets a set of criteria regarding naming and versioning. A module signature is created via the CreateModuleSignature method on IDebugHostSymbols. It can match the module name, and an optional range of version numbers for the module. Once such a signature is created, the client receives an IDebugHostModuleSignature interface which is defined as follows:

```
DECLARE_INTERFACE_(IDebugHostModuleSignature, IUnknown)
{
    STDMETHOD(IsMatch)(_In_ IDebugHostModule* pModule, _Out_ bool* isMatch) PURE;
}
```

IsMatch

The IsMatch method compares a particular module (as given by an IDebugHostModule symbol) against a signature, comparing the module name and version to the name and version range indicated in the signature. An indication of whether the given module symbol matches the signature is returned.

Type Signatures and Type Matching: IDebugHostTypeSignature

Type signatures represent a means to check whether a given type instance meets a set of criteria about the name of the type, the generic arguments to the type, and the module that the type is located within. A type signature is created via the CreateTypeSignature method on IDebugHostSymbols. Once such a signature is created, the client receives an IDebugHostTypeSignature interface which is defined as follows:

```
DECLARE_INTERFACE_(IDebugHostTypeSignature, IUnknown)
{
    STDMETHOD(GetHashCode)(_Out_ ULONG* hashCode) PURE;
    STDMETHOD(IsMatch)(_In_ IDebugHostType* type, _Out_ bool* isMatch, _COM_Outptr_opt_
IDebugHostSymbolEnumerator** wildcardMatches) PURE;
    STDMETHOD(CompareAgainst)(_In_ IDebugHostTypeSignature* typeSignature, _Out_ SignatureComparison* result)
PURE;
}
```

GetHashCode

The GetHashCode method returns a 32-bit hash code for the type signature. The debug host guarantees that there is synchronization in implementation between the hash code returned for type instances and the hash code

returned for type signatures. With the exception of a global match, if a type instance is capable of matching a type signature, both will have the same 32-bit hash code. This allows an initial rapid comparison and match between a type instance and a plethora of type signatures registered with the data model manager.

IsMatch

The IsMatch method returns an indication of whether a particular type instance matches the criteria specified in the type signature. If it does, an indication of this is returned as well as an enumerator which will indicate all of the specific portions of the type instance (as symbols) which matched wildcards in the type signature.

CompareAgainst

The CompareAgainst method compares the type signature to another type signature and returns how the two signatures compare. The comparison result which is returned is a member of the SignatureComparison enumeration which is defined as follows:

ENUMERANT	MEANING
Unrelated	There is no relationship between the two signatures or types being compared.
Ambiguous	One signature or type compares ambiguously against the other. For two type signatures, this means that there are potential type instances which could match either signature equally well. As an example, the two type signatures shown below are ambiguous. Signature 1: `std::pair<*, int>` Signature 2: `std::pair<int,*>` because the type instance `std::pair<int, int>` matches either one equally well (both have one concrete and one wildcard match).
LessSpecific	One signature or type is less specific than the other. Often, this means that the less specific signature has a wildcard where the more specific one has a concrete type. As an example, the first signature below is less specific than the second. Signature 1: `std::pair<*, int>` Signature 2: `std::pair<int, int>` because it has a wildcard (the `*`) where the second has a concrete type (int).
MoreSpecific	One signature or type is more specific than the other. Often, this means that the more specific signature has a concrete type where the less specific one has a wildcard. As an example, the first signature below is more specific than the second. Signature 1: `std::pair<int, int>` Signature 2: `std::pair<*, int>` because it has a concrete type (int) where the second has a wildcard (the `*`).
Identical	The two signatures or types are identical.

Related topics

This topic is part of a series which describes the interfaces accessible from C++, how to use them to build a C++ based debugger extension, and how to make use of other data model constructs (e.g.: JavaScript or NatVis) from a C++ data model extension.

Debugger Data Model C++ Overview

Debugger Data Model C++ Interfaces

Debugger Data Model C++ Objects

Debugger Data Model C++ Additional Interfaces

Debugger Data Model C++ Concepts

Debugger Data Model C++ Scripting

Debugger Data Model C++ Objects

10/23/2019 • 37 minutes to read • Edit Online

This topic describes how to use Debugger Data Model C++ Objects and how they can extend the capabilities of the debugger.

The Core Debugger Object Model

One of the most basic yet powerful things about the data model is that it standardizes the definition of what an object is and how one interacts with an object. The *IModelObject* interface encapsulates the notion of an object -- whether that object is an integer, a floating point value, a string, some complex type in the target address space of the debugger, or some debugger concept like the notion of a process or a module.

There are several different things that can be held in (or boxed into) an *IModelObject*:

- **Intrinsic Values** - An *IModelObject* can be a container for a number of basic types: 8, 16, 32, or 64 bit signed or unsigned integers, booleans, strings, errors, or the notion of empty.
- **Native Objects** - An *IModelObject* can represent a complex type (as defined by the debugger's type system) within the address space of whatever the debugger is targeting
- **Synthetic Objects** - An *IModelObject* can be a dynamic object -- a dictionary if you will: a collection of key / value / metadata tuples and a set of **concepts** which define behaviors that aren't simply represented by key / value pairs.
- **Properties** - An *IModelObject* can represent a property: something whose value can be retrieved or altered with a method call. A property within an *IModelObject* is effectively an *IModelPropertyAccessor* interface boxed into an *IModelObject*
- **Methods** - An *IModelObject* can represent a method: something you can call with a set of arguments and get a return value. A method within an *IModelObject* is effectively an *IModelMethod* interface boxed into an *IModelObject*

Extensibility Within The Object Model

An *IModelObject* is not an object in isolation. In addition to representing one of the types of objects shown above, each object has the notion of a chain of parent data models. This chain behaves much like a JavaScript prototype chain. Instead of a linear chain of prototypes like JavaScript has, each data model object defines a linear chain of **parent models**. Each of those parent models in turn has another linear chain of its own set of parents. In essence, each object is an aggregation of the capabilities (properties, etc...) of both itself and every object in this tree. When a specific property is queried, if the object it is queried on does not support that property, the query is passed in linear order to each parent in turn. This creates a behavior where the search for a property is resolved by a depth-first search of the aggregate tree.

Extensibility within this object model is very simple given this notion that every object is an aggregate of itself and the tree of parent models. An extension can come in and add itself into the list of parent models for another object. Doing this **extends** the object. In this manner, it is possible to add capabilities onto anything: a particular instance of an object or value, a native type, the debugger's concept of what a process or thread is, or even the notion of "all iterable objects".

Context, Context, and Context: The this Pointer, The Address Space, and Implementation Private Data

There are three notions of **context** which are necessary to understand within the context of the object model.

Context: The this Pointer

Since a given property or method may be implemented at any level of the data model tree, it is necessary for the implementation of the method or property to be able to access the original object (what you might call the **this** pointer in C++ or the **this** object in JavaScript. That instance object is passed to a variety of methods as the first argument called **context** in the methods described.

Context: The Address Space

It is important to note that unlike prior extension models where **context** (the target, process, thread you are looking at) is a UI concept with all APIs relative to the current UI state, data model interfaces typically take this context either explicitly or implicitly as an *IDebugHostContext* interface. Each *IModelObject* within the data model carries this type of context information along with it and can propagate that context to objects it returns. This means that when you read a native value or a key value out of an *IModelObject*, it will read out of the target and process where the object was originally acquired from.

There is an explicit constant value, **USE_CURRENT_HOST_CONTEXT**, that can be passed to methods which take an *IDebugHostContext* argument. This value indicates that the context should indeed be the current UI state of the debugger. This notion does, however, need to be explicit.

Context: Implementation Private Data

Remember that each object in the data model is actually an aggregate of the object instance and the tree of parent models which are attached. Each of those parent models (which can be linked in the chains of many different objects) can associate private implementation data with any instance object. Each *IModelObject* which is created conceptually has a hash table that maps from a particular parent model to private instance data defined by an *IUnknown* interface. This allows a parent model to cache information on every instance or have otherwise arbitrary data associated.

This type of context is accessed through the *GetContextForDataModel* and *SetContextForDataModel* methods on *IModelObject*.

The Core Debugger Object Interface: **IModelObject**

The *IModelObject* interface is defined as follows:

```
DECLARE_INTERFACE_(IModelObject, IUnknown)
{
    STDMETHOD(QueryInterface)(_In_ REFIID iid, _COM_Outptr_ PVOID* iface);
    STDMETHOD_(ULONG, AddRef)();
    STDMETHOD_(ULONG, Release)() PURE;
    STDMETHOD(GetContext)(_COM_Outptr_result_maybenull_ IDebugHostContext** context) PURE;
    STDMETHOD(GetKind)(_Out_ ModelObjectKind *kind) PURE;
    STDMETHOD(GetIntrinsicValue)(_Out_ VARIANT* intrinsicData);
    STDMETHOD(GetIntrinsicValueAs)(_In_ VARTYPE vt, _Out_ VARIANT* intrinsicData) PURE;
    STDMETHOD(GetKeyValue)
(_In_ PCWSTR key, _COM_Errorptr_opt_ IModelObject** object, _COM_Outptr_opt_result_maybenull_ IKeyStore** meta
data) PURE;
    STDMETHOD(SetKeyValue)(_In_ PCWSTR key, _In_opt_ IModelObject* object) PURE;
    STDMETHOD(EnumerateKeyValues)(_COM_Outptr_ IKeyEnumerator** enumerator) PURE;
    STDMETHOD(GetRawValue)
(_In_ SymbolKind kind, _In_ PCWSTR name, _In_ ULONG searchFlags, _COM_Errorptr_ IModelObject** object) PURE;
    STDMETHOD(EnumerateRawValues)
(_In_ SymbolKind kind, _In_ ULONG searchFlags, _COM_Outptr_ IRawEnumerator** enumerator) PURE;
    STDMETHOD(Dereference)(_COM_Errorptr_ IModelObject** object) PURE;
    STDMETHOD(TryCastToRuntimeType)(_COM_Errorptr_ IModelObject** runtimeTypedObject) PURE;
    STDMETHOD(GetConcept)
(_In_ REFIID conceptId, _COM_Outptr_ IUnknown** conceptInterface, _COM_Outptr_opt_result_maybenull_ IKeyStore*
* conceptMetadata) PURE;
    STDMETHOD(GetLocation)(_Out_ Location* location) PURE;
    STDMETHOD(GetTypeInfo)(_Out_ IDebugHostType** type) PURE;
    STDMETHOD(GetTargetInfo)(_Out_ Location* location, _Out_ IDebugHostType** type) PURE;
    STDMETHOD(GetNumberOfParentModels)(_Out_ ULONG64* numModels) PURE;
    STDMETHOD(GetParentModel)
(_In_ ULONG64 i, _COM_Outptr_ IModelObject **model, _COM_Outptr_result_maybenull_ IModelObject **contextObject
) PURE;
    STDMETHOD(AddParentModel)
(_In_ IModelObject* model, _In_opt_ IModelObject* contextObject, _In_ bool override) PURE;
    STDMETHOD(RemoveParentModel)(_In_ IModelObject* model) PURE;
    STDMETHOD(GetKey)
(_In_ PCWSTR key, _COM_Errorptr_opt_ IModelObject** object, _COM_Outptr_opt_result_maybenull_ IKeyStore** meta
data) PURE;
    STDMETHOD(GetKeyReference)
(_In_ PCWSTR key, _COM_Errorptr_opt_ IModelObject** objectReference, _COM_Outptr_opt_result_maybenull_ IKeySto
re** metadata) PURE;
    STDMETHOD(SetKey)(_In_ PCWSTR key, _In_opt_ IModelObject* object, _In_opt_ IKeyStore* metadata) PURE;
    STDMETHOD(ClearKeys)() PURE;
    STDMETHOD(EnumerateKeys)(_COM_Outptr_ IKeyEnumerator** enumerator) PURE;
    STDMETHOD(EnumerateKeyReferences)(_COM_Outptr_ IKeyEnumerator** enumerator) PURE;
    STDMETHOD(SetConcept)
(_In_ REFIID conceptId, _In_ IUnknown* conceptInterface, _In_opt_ IKeyStore* conceptMetadata) PURE;
    STDMETHOD(ClearConcepts)() PURE;
    STDMETHOD(GetRawReference)
(_In_ SymbolKind kind, _In_ PCWSTR name, _In_ ULONG searchFlags, _COM_Errorptr_ IModelObject** object) PURE;
    STDMETHOD(EnumerateRawReferences)
(_In_ SymbolKind kind, _In_ ULONG searchFlags, _COM_Outptr_ IRawEnumerator** enumerator) PURE;
    STDMETHOD(SetContextForDataModel)(_In_ IModelObject* dataModelObject, _In_ IUnknown* context) PURE;
    STDMETHOD(GetContextForDataModel)(_In_ IModelObject* dataModelObject, _Out_ IUnknown** context) PURE;
    STDMETHOD(Compare)
(_In_ IModelObject* other, _COM_Outptr_opt_result_maybenull_ IModelObject **ppResult) PURE;
    STDMETHOD(IsEqualTo)(_In_ IModelObject* other, _Out_ bool* equal) PURE;
}
```

Basic Methods

The following are general methods applicable to any kind of object represented by an IModelObject.

```
STDMETHOD(GetKind)(_Out_ ModelObjectKind *kind) PURE;
STDMETHOD(GetContext)(_COM_Outptr_result_maybenull_ IDebugHostContext** context) PURE;
STDMETHOD(GetIntrinsicValue)(_Out_ VARIANT* intrinsicData);
STDMETHOD(GetIntrinsicValueAs)(_In_ VARTYPE vt, _Out_ VARIANT* intrinsicData) PURE;
STDMETHOD(Compare)(_In_ IModelObject* other, _COM_Outptr_opt_result_maybenull_ IModelObject **ppResult) PURE;
STDMETHOD(IsEqualTo)(_In_ IModelObject* other, _Out_ bool* equal) PURE;
STDMETHOD(Dereference)(_COM_Errorptr_ IModelObject** object) PURE;
```

GetKind

The GetKind method returns what kind of object is boxed inside the IModelObject.

GetContext

The GetContext method returns the host context that is associated with the object.

GetIntrinsicValue

The GetIntrinsicValue method returns the thing which is boxed inside an IModelObject. This method may only legally be called on IModelObject interfaces which represent a boxed intrinsic or a particular interface which is boxed. It cannot be called on native objects, no value objects, synthetic objects, and reference objects. The GetIntrinsicValueAs method behaves much as the GetIntrinsicValue method excepting that it converts the value to the specified variant type. If the conversion cannot be performed, the method returns an error.

IsEqualTo

The IsEqualTo method compares two model objects and returns whether they are equal in value. For object which have an ordering, this method returning true is equivalent to the Compare method returning 0. For objects that have no ordering but are equatable, the Compare method will fail, but this will not. The meaning of a value based comparison is defined by the type of object. At present, this is only defined for intrinsic types and error objects. There is no current data model concept for equatability.

Dereference

The Dereference method dereferences an object. This method can be used to dereference a data model based reference (ObjectTargetObjectReference, ObjectKeyReference) or a native language reference (a pointer or a language reference). It is important to note that this method removes a single level of reference semantics on the object. It is entirely possible to, for instance, have a data model reference to a language reference. In such a case, calling the Dereference method the first time would remove the data model reference and leave the language reference. Calling Dereference on that resulting object would subsequently remove the language reference and return the native value under that reference.

Key Manipulation Methods

Any synthetic object which is a dictionary of key, value, and metadata tuples has a series of methods to manipulate those keys, values, and the metadata associated with them.

The value based forms of the APIs are:

```
STDMETHOD(GetKeyValue)(_In_ PCWSTR key, _COM_Errorptr_opt_ IModelObject** object,
_COM_Outptr_opt_result_maybenull_ IKeyStore** metadata) PURE;
STDMETHOD(SetKeyValue)(_In_ PCWSTR key, _In_opt_ IModelObject* object) PURE;
STDMETHOD(EnumerateKeyValues)(_COM_Outptr_ IKeyEnumerator** enumerator) PURE;
The key based forms of the APIs (including those used for key creation) are:
STDMETHOD(GetKey)(_In_ PCWSTR key, _COM_Errorptr_opt_ IModelObject** object, _COM_Outptr_opt_result_maybenull_
IKeyStore** metadata) PURE;
STDMETHOD(SetKey)(_In_ PCWSTR key, _In_opt_ IModelObject* object, _In_opt_ IKeyStore* metadata) PURE;
STDMETHOD(EnumerateKeys)(_COM_Outptr_ IKeyEnumerator** enumerator) PURE;
STDMETHOD(ClearKeys)() PURE;
```

The reference based forms of the APIs are:

```
STDMETHOD(GetKeyReference)(_In_ PCWSTR key, _COM_Errorptr_opt_ IModelObject** objectReference,
_COM_Outptr_opt_result_maybenull_ IKeyStore** metadata) PURE;
STDMETHOD(EnumerateKeyReferences)(_COM_Outptr_ IKeyEnumerator** enumerator) PURE;
```

GetKeyValue

The GetKeyValue method is the first method a client will turn to in order to get the value of (and the metadata associated with) a given key by name. If the key is a property accessor -- that is it's value as an IModelObject which is a boxed IModelPropertyAccessor, the GetKeyValue method will automatically call the property accessor's GetValue method in order to retrieve the actual value.

SetKeyValue

The SetKeyValue method is the first method a client will turn to in order to set the value of a key. This method cannot be used to create a new key on an object. It will only set the value of an existing key. Note that many keys are read-only (e.g.: they are implemented by a property accessor which returns E_NOT_IMPL from it's SetValue method). This method will fail when called on a read only key.

EnumerateKeyValues

The EnumerateKeyValues method is the first method a client will turn to in order to enumerate all of the keys on an object (this includes all keys implemented anywhere in the tree of parent models). It is important to note that EnumerateKeyValues will enumerate any keys defined by duplicate names in the object tree; however -- methods like GetKeyValue and SetKeyValue will only manipulate the first instance of a key with the given name as discovered by the depth-first-traversal.

GetKey

The GetKey method will get the value of (and the metadata associated with) a given key by name. Most clients should utilize the GetKeyValue method instead. If the key is a property accessor, calling this method will return the property accessor (an IModelPropertyAccessor interface) boxed into an IModelObject. Unlike, GetKeyValue, this method will not automatically resolve the underlying value of the key by calling the GetValue method. That responsibility is the caller's.

SetKey

The SetKey method is the method a client will turn to in order to create a key on an object (and potentially associate metadata with the created key). If a given object already has a key with the given name, one of two behaviors will occur. If the key is on the instance given by this, the value of that key will be replaced as if the original key did not exist. If, on the other hand, the key is in the chain of parent data models of the instance given by this, a new key with the given name will be created on the instance given by this. This would, in effect, cause the object to have two keys of the same name (similar to a derived class shadowing a member of the same name as a base class).

EnumerateKeys

The EnumerateKeys method behaves similar to the EnumerateKeyValues method excepting that it does not automatically resolve property accessors on the object. This means that if the value of a key is a property accessor, the EnumerateKeys method will return the property accessor (an IModelPropertyAccessorInterface) boxed into an IModelObject rather than automatically calling the GetValue method. This is similar to the difference between GetKey and GetKeyValue.

ClearKeys

The ClearKeys method removes all keys and their associated values and metadata from the instance of the object specified by this. This method has no effect on parent models attached to the particular object instance.

GetKeyReference

The GetKeyReference method will search for a key of the given name on the object (or its parent model chain) and return a reference to that key given by an IModelKeyReference interface boxed into an IModelObject. That reference can subsequently be used to get or set the value of the key.

EnumerateKeyReferences

The EnumerateKeyReferences method behaves similar to the EnumerateKeyValues method excepting that it returns references to the keys it enumerates (given by an IModelKeyReference interface boxed into an IModelObject) instead of the value of the key. Such references can be used to get or set the underlying value of the keys.

Concept Manipulation Methods

In addition to a model object being a dictionary of key/value/metadata tuples, it is also a container of concepts. A concept is something abstract that can be performed on or by an object. Concepts are, in essence, a dynamic store of interfaces that an object supports. A number of concepts are defined by the data model today:

CONCEPT INTERFACE	DESCRIPTION
IDataModelConcept	The concept is a parent model. If this model is automatically attached to a native type via a registered type signature, the InitializeObject method will automatically be called every time a new object of such type is instantiated.
IStringDisplayableConcept	The object can be converted to a string for display purposes.
IIterableConcept	The object is a container and can be iterated.
IIndexableConcept	The object is a container and can be indexed (accessed via random access) in one or more dimensions.
IPreferredRuntimeTypeConcept	The object understands more about types derived from it than the underlying type system is capable of providing and would like to handle its own conversions from static to runtime type.
IDynamicKeyProviderConcept	The object is a dynamic provider of keys and wishes to take over all key queries from the core data model. This interface is typically used as a bridge to dynamic languages such as JavaScript.
IDynamicConceptProviderConcept	The object is a dynamic provider of concepts and wishes to take over all concept queries from the core data model. This interface is typically used as a bridge to dynamic languages such as JavaScript.

The following methods on IModelObject are utilized to manipulate the concepts that an object supports.

```
STDMETHOD(GetConcept)(_In_ REFIID conceptId, _COM_Outptr_ IUnknown** conceptInterface,
_COM_Outptr_opt_result_maybenull_ IKeyStore** conceptMetadata) PURE;
STDMETHOD(SetConcept)(_In_ REFIID conceptId, _In_ IUnknown* conceptInterface, _In_opt_ IKeyStore*
conceptMetadata) PURE;
STDMETHOD(ClearConcepts)() PURE;
```

GetConcept

The GetConcept method will search for a concept on the object (or its parent model chain) and return an interface pointer to the concept interface. The behavior and methods on a concept interface are specific to each concept. It is important to note, however, that many of the concept interfaces require the caller to explicitly pass the context object (or what one might traditionally call the this pointer). It is important to ensure that the correct context object is passed to every concept interface.

SetConcept

The SetConcept method will place a specified concept on the object instance specified by the this pointer. If a parent model attached to the object instance specified by this also supports the concept, the implementation in the instance will override that in the parent model.

ClearConcepts

The ClearConcepts method will remove all concepts from the instance of the object specified by this.

Native Object Methods

While many model objects refer to intrinsics (e.g.: integers, strings) or synthetic constructs (a dictionary of key/value/metadata tuples and concepts), a model object may also refer to a native construct (e.g.: a user defined type in the address space of the debug target). The IModelObject interface has a series of methods on it which access information about such native objects. Those methods are:

```
STDMETHOD(GetRawValue)(_In_ SymbolKind kind, _In_ PCWSTR name, _In_ ULONG searchFlags, _COM_Errorptr_
IModelObject** object) PURE;
STDMETHOD(EnumerateRawValues)(_In_ SymbolKind kind, _In_ ULONG searchFlags, _COM_Outptr_ IRawEnumerator**
enumerator) PURE;
STDMETHOD(TryCastToRuntimeType)(_COM_Errorptr_ IModelObject** runtimeTypedObject) PURE;
STDMETHOD(GetLocation)(_Out_ Location* location) PURE;
STDMETHOD(GetTypeInfo)(_Out_ IDebugHostType** type) PURE;
STDMETHOD(GetTargetInfo)(_Out_ Location* location, _Out_ IDebugHostType** type) PURE;
STDMETHOD(GetRawReference)(_In_ SymbolKind kind, _In_ PCWSTR name, _In_ ULONG searchFlags, _COM_Errorptr_
IModelObject** object) PURE;
STDMETHOD(EnumerateRawReferences)(_In_ SymbolKind kind, _In_ ULONG searchFlags, _COM_Outptr_ IRawEnumerator**
enumerator) PURE;
```

GetRawValue

The GetRawValue method finds a native construct within the given object. Such a construct may be a field, a base class, a field in a base class, a member function, etc...

EnumerateRawValues

The EnumerateRawValues method enumerates all native children (e.g.: fields, base classes, etc...) of the given object.

TryCastToRuntimeType

The TryCastToRuntimeType method will ask the debug host to perform an analysis and determine the actual runtime type (e.g.: most derived class) of the given object. The exact analysis utilized is specific to the debug host and may include RTTI (C++ run time type information), examination of the V-Table(virtual function table) structure of the object, or any other means that the host can use to reliably determine dynamic/runtime type from the static type. Failure to convert to a runtime type does not mean that this method call will fail. In such cases, the method will return the given object (the this pointer) in the output argument.

GetLocation

The GetLocation method will return the location of the native object. While such a location is typically a virtual address within the address space of the debug target, it is not necessarily so. The location returned by this method is an abstract location that may be a virtual address, may indicate placement within a register or sub-register, or

may indicate some other arbitrary address space as defined by the debug host. If the HostDefined field of the resulting Location object is 0, it indicates that the location is actually a virtual address. Such virtual address may be retrieved by examining the Offset field of the resulting location. Any non-zero value of the HostDefined field indicates an alternate address space where the Offset field is the offset within that address space. The exact meaning of non-zero HostDefined values here are private to the debug host.

GetTypeInfo

The GetTypeInfo method will return the native type of the given object. If the object does not have native type information associated with it (e.g.: it is an intrinsic, etc...), the call will still succeed but will return null.

GetTargetInfo

The GetTargetInfo method is effectively a combination of the GetLocation and GetTypeInfo methods returning both the abstract location as well as native type of the given object.

GetRawReference

The GetRawReference method finds a native construct within the given object and returns a reference to it. Such a construct may be a field, a base class, a field in a base class, a member function, etc... It is important to distinguish the reference returned here (an object of the type ObjectTargetObjectReference) from a language reference (e.g.: a C++ & or && style reference).

EnumerateRawReferences

The EnumerateRawReferences method enumerates references to all native children (e.g.: fields, base classes, etc...) of the given object.

Extensibility Methods

As described earlier, a model object behaves very similar to a JavaScript object and its prototype chain. In addition to the instance represented by a given IModelObject interface, there may be an arbitrary number of parent models attached to the object (each of which may, in turn, have an arbitrary number of parent models attached to them). This is the primary means for extensibility within the data model. If a given property or concept cannot be located within a given instance, a depth-first search of the object tree (defined by parent models) rooted at the instance is performed.

The following methods manipulate the chain of parent models associated with a given IModelObject instance:

```
STDMETHOD(GetNumberOfParentModels)(_Out_ ULONG64* numModels) PURE;
STDMETHOD(GetParentModel)(_In_ ULONG64 i, _COM_Outptr_ IModelObject **model, _COM_Outptr_result_maybenull_
IModelObject **contextObject) PURE;
STDMETHOD(AddParentModel)(_In_ IModelObject* model, _In_opt_ IModelObject* contextObject, _In_ bool override)
PURE;
STDMETHOD(RemoveParentModel)(_In_ IModelObject* model) PURE;
STDMETHOD(SetContextForDataModel)(_In_ IModelObject* dataModelObject, _In_ IUnknown* context) PURE;
STDMETHOD(GetContextForDataModel)(_In_ IModelObject* dataModelObject, _Out_ IUnknown** context) PURE;
```

GetNumberOfParentModels

The GetNumberOfParentModels method returns the number of parent models which are attached to the given object instance. Parent models are searched for properties depth-first in the linear ordering of the parent model chain.

GetParentModel

The GetParentModel method returns the i-th parent model in the parent model chain of the given object. Parent models are searched for a property or concept in the linear order they are added or enumerated. The parent model with index i of zero is searched (hierarchically) before the parent model with index i + 1.

AddParentModel

The AddParentModel method adds a new parent model to the given object. Such a model may be added at the end of the search chain (the override argument is specified as false) or at the front of the search chain (the override argument is specified as true). In addition, each parent model may optionally adjust the context (the semantic this pointer) for any property or concept on the given parent (or anyone in its parent hierarchy). Context adjustment is seldom used but allows for some powerful concepts like object embedding, constructing namespaces, etc...

RemoveParentModel

The RemoveParentModel will remove a specified parent model from the parent search chain of the given object.

SetContextForDataModel

The SetContextForDataModel method is used by the implementation of a data model to place implementation data on instance objects. Conceptually, each IModelObject (call this the instance for simplicity) contains a hash table of state. The hash table is indexed by another IModelObject (call this the data model for simplicity) which is in the parent model hierarchy of the instance. The value contained in this hash is a set of reference counted state information represented by an IUnknown instance. Once the data model sets this state on the instance it can store arbitrary implementation data which can be retrieved during things like property getters.

GetContextForDataModel

The GetContextForDataModel method is used to retrieve context information which was set up with a prior call to SetContextForDataModel. This retrieves state information which was set on an instance object by a data model further up in the instance object's parent model hierarchy. For more details about this context/state and its meaning, see the documentation for SetContextForDataModel.

Debugger Data Model Core Object Types

An object in the data model is similar to the notion of Object in .NET. It is the generic container into which construct that the data model understands can be boxed. In addition to native objects and synthetic (dynamic) objects, there are a series of core object types which can be placed (or boxed) into the container of an IModelObject. The container in which most of these values are placed is a standard COM/OLE VARIANT with a number of additional restrictions placed upon what that VARIANT can contain. The most basic types of these are:

- 8-bit unsigned and signed values (VT_UI1, VT_I1)
- 16-bit unsigned and signed values (VT_UI2, VT_UI2)
- 32-bit unsigned and signed values (VT_UI4, VT_I4)
- 64-bit unsigned and signed values (VT_UI8, VT_I8)
- Single and double precision floating point values (VT_R4, VT_R8)
- Strings (VT_BSTR)
- Booleans (VT_BOOL)

In addition to these basic types, a number of core data model objects are placed into IModelObject defined by VT_UNKNOWN where the stored IUnknown is guaranteed to implement a specific interface. These types are:

- Property accessors (IModelPropertyAccessor)
- Method objects (IModelMethod)
- Key reference objects (IModelKeyReference or IModelKeyReference2)
- Context objects (IDebugModelHostContext)

Property Accessors: *IModelPropertyAccessor*

A property accessor in the data model is an implementation of the IModelPropertyAccessor interface which is

boxed into an IModelObject. The model object will return a kind of ObjectPropertyAccessor when queried and the intrinsic value is a VT_UNKNOWN which is guaranteed to be queryable for IModelPropertyAccessor. In process, it is guaranteed to be statically castable to IModelPropertyAccessor.

A property accessor is an indirect way to get a method call for getting and setting a key value in the data model. If a given key's value is a property accessor, the GetKeyValue and SetKeyValue methods will automatically notice this and call the property accessor's underlying GetValue or SetValue methods as appropriate.

The IModelPropertyAccessor interface is defined as follows:

```
DECLARE_INTERFACE_(IModelPropertyAccessor, IUnknown)
{
    STDMETHOD(GetValue)(_In_ PCWSTR key, _In_opt_ IModelObject* contextObject, _COM_Outptr_ IModelObject**
value) PURE;
    STDMETHOD(SetValue)(_In_ PCWSTR key, _In_opt_ IModelObject* contextObject, _In_ IModelObject* value) PURE;
}
```

GetValue

The GetValue method is the getter for the property accessor. It is called whenever a client wishes to fetch the underlying value of the property. Note that any caller which directly gets a property accessor is responsible for passing the key name and accurate instance object (this pointer) to the property accessor's GetValue method.

SetValue

The SetValue method is the setter for the property accessor. It is called whenever a client wishes to assign a value to the underlying property. Many properties are read-only. In such cases, calling the SetValue method will return E_NOTIMPL. Note that any caller which directly gets a property accessor is responsible for passing the key name and accurate instance object (this pointer) to the property accessor's SetValue method.

Methods: *IModelMethod*

A method in the data model is an implementation of the IModelMethod interface which is boxed into an IModelObject. The model object will return a kind of ObjectMethod when queried and the intrinsic value is a VT_UNKNOWN which is guaranteed to be queryable for IModelMethod. In process, it is guaranteed to be statically castable to IModelMethod. All methods in the data model are dynamic in nature. They take as input a set of 0 or more arguments and return a single output value. There is no overload resolution and no metadata about parameter names, types, or expectations.

The IModelMethod interface is defined as follows:

```
DECLARE_INTERFACE_(IModelMethod, IUnknown)
{
    STDMETHOD(Call)(_In_opt_ IModelObject *pContextObject, _In_ ULONG64 argCount, _In_reads_(argCount)
IModelObject **ppArguments, _COM_Errorptr_ IModelObject **ppResult, _COM_Outptr_opt_result_maybenull_
IKeyStore **ppMetadata) PURE;
}
```

Call

The Call method is the way in which any method defined in the data model is invoked. The caller is responsible for passing an accurate instance object (this pointer) and an arbitrary set of arguments. The result of the method and any optional metadata associated with that result is returned. Methods which do not logically return a value still must return a valid IModelObject. In such a case, the IModelObject is a boxed no value. In the event a method fails, it may return optional extended error information in the input argument (even if the returned HRESULT is a failure). It is imperative that callers check for this.

Key References: *IModelKeyReference or IModelKeyReference2*

A key reference is, in essence, a handle to a key on a particular object. A client can retrieve such handle via methods such as GetKeyReference and use the handle later to get or set the value of the key without necessarily holding onto the original object. This type of object is an implementation of the IModelKeyReference or IModelKeyReference2 interface which is boxed into an IModelObject. The model object will return a kind of ObjectKeyReference when queried and then intrinsic value is a VT_UNKNOWN which is guaranteed to be queryable for IModelKeyReference. In process, it is guaranteed to be statically castable to IModelKeyReference.

The key reference interface is defined as follows:

```
DECLARE_INTERFACE_(IModelKeyReference2, IModelKeyReference)
{
    STDMETHOD(GetKeyName)(_Out_ BSTR* keyName) PURE;
    STDMETHOD(GetOriginalObject)(_COM_Outptr_ IModelObject** originalObject) PURE;
    STDMETHOD(GetContextObject)(_COM_Outptr_ IModelObject** containingObject) PURE;
    STDMETHOD(GetKey)(_COM_Errorptr_opt_ IModelObject** object, _COM_Outptr_opt_result_maybenull_ IKeyStore**
metadata) PURE;
    STDMETHOD(GetKeyValue)(_COM_Errorptr_opt_ IModelObject** object, _COM_Outptr_opt_result_maybenull_
IKeyStore** metadata) PURE;
    STDMETHOD(SetKey)(_In_opt_ IModelObject* object, _In_opt_ IKeyStore* metadata) PURE;
    STDMETHOD(SetKeyValue)(_In_ IModelObject* object) PURE;
    STDMETHOD(OverrideContextObject)(_In_ IModelObject* newContextObject) PURE;
}
```

GetKeyName

The GetKeyName method returns the name of the key to which this key reference is a handle. The returned string is a standard BSTR and must be freed via a call to SysFreeString.

GetOriginalObject

The GetOriginalObject method returns the instance object from which the key reference was created. Note that the key may itself be on a parent model of the instance object.

GetContextObject

The GetContextObject method returns the context (this pointer) which will be passed to a property accessor's GetValue or SetValue method if the key in question refers to a property accessor. The context object returned here may or may not be the same as the original object fetched from GetOriginalObject. If a key is on a parent model and there is a context adjustor associated with that parent model, the original object is the instance object on which GetKeyReference or EnumerateKeyReferences was called. The context object would be whatever comes out of the final context adjustor between the original object and the parent model containing the key to which this key reference is a handle. If there are no context adjustors, the original object and the context object are identical.

GetKey

The GetKey method on a key reference behaves as the GetKey method on IModelObject would. It returns the value of the underlying key and any metadata associated with the key. If the value of the key happens to be a property accessor, this will return the property accessor (IModelPropertyAccessor) boxed into an IModelObject. This method will not call the underlying GetValue or SetValue methods on the property accessor.

GetKeyValue

The GetKeyValue method on a key reference behaves as the GetKeyValue method on IModelObject would. It returns the value of the underlying key and any metadata associated with the key. If the value of the key happens to be a property accessor, this will call the underlying GetValue method on the property accessor automatically.

SetKey

The SetKey method on a key reference behaves as the SetKey method on IModelObject would. It will assign the value of the key. If the original key was a property accessor, this will replace the property accessor. It will not call

the SetValue method on the property accessor.

SetKeyValue

The SetKeyValue method on a key reference behaves as the SetKeyValue method on IModelObject would. It will assign the value of the key. If the original key was a property accessor, this will call the underlying SetValue method on the property accessor rather than replacing the property accessor itself.

OverrideContextObject

The OverrideContextObject method (only present on IModelKeyReference2) is an advanced method which is used to permanently alter the context object which this key reference will pass to any underlying property accessor's GetValue or SetValue methods. The object passed to this method will also be returned from a call to GetContextObject. This method can be used by script providers to replicate certain dynamic language behaviors. Most clients should not call this method.

Context Objects: *IDebugHostContext*

Context objects are opaque blobs of information that the debug host (in cooperation with the data model) associates with every object. It may include things such as the process context or address space the information comes from, etc... A context object is an implementation of IDebugHostContext boxed within an IModelObject. Note that IDebugHostContext is a host defined interface. A client will never implement this interface.

For more information about context objects, see Debugger Data Model C++ Host Interfaces in Debugger Data Model C++ Interfaces.

The Data Model Manager

The core interface to the data model manager, IDataModelManager2 (or the earlier IDataModelManager) is defined as follows:

```
DECLARE_INTERFACE_(IDataModelManager2, IDataModelManager)
{
    //
    // IDataModelManager:
    //
    STDMETHOD(Close)() PURE;
    STDMETHOD(CreateNoValue)(_Out_ IModelObject** object) PURE;
    STDMETHOD(CreateErrorObject)(_In_ HRESULT hrError, _In_opt_ PCWSTR pwszMessage, _COM_Outptr_
IModelObject** object) PURE;
    STDMETHOD(CreateTypedObject)(_In_opt_ IDebugHostContext* context, _In_ Location objectLocation, _In_
IDebugHostType* objectType, _COM_Errorptr_ IModelObject** object) PURE;
    STDMETHOD(CreateTypedObjectReference)(_In_opt_ IDebugHostContext* context, _In_ Location objectLocation,
_In_ IDebugHostType* objectType, _COM_Errorptr_ IModelObject** object) PURE;
    STDMETHOD(CreateSyntheticObject)(_In_opt_ IDebugHostContext* context, _COM_Outptr_ IModelObject** object)
PURE;
    STDMETHOD(CreateDataModelObject)(_In_ IDataModelConcept* dataModel, _COM_Outptr_ IModelObject** object)
PURE;
    STDMETHOD(CreateIntrinsicObject)(_In_ ModelObjectKind objectKind, _In_ VARIANT* intrinsicData,
_COM_Outptr_ IModelObject** object) PURE;
    STDMETHOD(CreateTypedIntrinsicObject)(_In_ VARIANT* intrinsicData, _In_ IDebugHostType* type, _COM_Outptr_
IModelObject** object) PURE;
    STDMETHOD(GetModelForTypeSignature)(_In_ IDebugHostTypeSignature* typeSignature, _Out_ IModelObject**
dataModel) PURE;
    STDMETHOD(GetModelForType)(_In_ IDebugHostType* type, _COM_Outptr_ IModelObject** dataModel,
_COM_Outptr_opt_ IDebugHostTypeSignature** typeSignature, _COM_Outptr_opt_ IDebugHostSymbolEnumerator**
wildcardMatches) PURE;
    STDMETHOD(RegisterModelForTypeSignature)(_In_ IDebugHostTypeSignature* typeSignature, _In_ IModelObject*
dataModel) PURE;
    STDMETHOD(UnregisterModelForTypeSignature)(_In_ IModelObject* dataModel, _In_opt_ IDebugHostTypeSignature*
typeSignature) PURE;
    STDMETHOD(RegisterExtensionForTypeSignature)(_In_ IDebugHostTypeSignature* typeSignature, _In_
IModelObject* dataModel) PURE;
    STDMETHOD(UnregisterExtensionForTypeSignature)(_In_ IModelObject* dataModel, _In_opt_
IDebugHostTypeSignature* typeSignature) PURE;
    STDMETHOD(CreateMetadataStore)(_In_opt_ IKeyStore* parentStore, _COM_Outptr_ IKeyStore** metadataStore)
PURE;
    STDMETHOD(GetRootNamespace)(_COM_Outptr_ IModelObject** rootNamespace) PURE;
    STDMETHOD(RegisterNamedModel)(_In_ PCWSTR modelName, _In_ IModelObject *modeObject) PURE;
    STDMETHOD(UnregisterNamedModel)(_In_ PCWSTR modelName) PURE;
    STDMETHOD(AcquireNamedModel)(_In_ PCWSTR modelName, _COM_Outptr_ IModelObject **modelObject) PURE;
    //
    // IDataModelManager2:
    //
    STDMETHOD(AcquireSubNamespace)(_In_ PCWSTR modelName, _In_ PCWSTR subNamespaceModelName, _In_ PCWSTR
accessName, _In_opt_ IKeyStore *metadata, _COM_Outptr_ IModelObject **namespaceModelObject) PURE;
    STDMETHOD(CreateTypedIntrinsicObjectEx)(_In_opt_ IDebugHostContext* context, _In_ VARIANT* intrinsicData,
_In_ IDebugHostType* type, _COM_Outptr_ IModelObject** object) PURE;
}
```

Management Methods

The following set of methods is utilized by the application (e.g.: debugger) hosting the data model.

```
STDMETHOD(Close)() PURE;
```

Close

The Close method is called on the data model manager by an application (e.g.: debugger) hosting the data model in order to start the shutdown process of the data model manager. A host of the data model which does not the Close method prior to releasing its final reference on the data model manager may cause undefined behavior including, but not limited to, significant leaks of the management infrastructure for the data model.

Object Creation / Boxing Methods

The following set of methods is used to create new objects or to box values into an IModelObject -- the core interface of the data model.

```
STDMETHOD(CreateNoValue)(_Out_ IModelObject** object) PURE;
STDMETHOD(CreateErrorObject)(_In_ HRESULT hrError, _In_opt_ PCWSTR pwszMessage, _COM_Outptr_ IModelObject**
object) PURE;
STDMETHOD(CreateTypedObject)(_In_opt_ IDebugHostContext* context, _In_ Location objectLocation, _In_
IDebugHostType* objectType, _COM_Errorptr_ IModelObject** object) PURE;
STDMETHOD(CreateTypedObjectReference)(_In_opt_ IDebugHostContext* context, _In_ Location objectLocation, _In_
IDebugHostType* objectType, _COM_Errorptr_ IModelObject** object) PURE;
STDMETHOD(CreateSyntheticObject)(_In_opt_ IDebugHostContext* context, _COM_Outptr_ IModelObject** object)
PURE;
STDMETHOD(CreateDataModelObject)(_In_ IDataModelConcept* dataModel, _COM_Outptr_ IModelObject** object) PURE;
STDMETHOD(CreateIntrinsicObject)(_In_ ModelObjectKind objectKind, _In_ VARIANT* intrinsicData, _COM_Outptr_
IModelObject** object) PURE;
STDMETHOD(CreateTypedIntrinsicObject)(_In_ VARIANT* intrinsicData, _In_ IDebugHostType* type, _COM_Outptr_
IModelObject** object) PURE;
STDMETHOD(CreateMetadataStore)(_In_opt_ IKeyStore* parentStore, _COM_Outptr_ IKeyStore** metadataStore) PURE;
STDMETHOD(CreateTypedIntrinsicObjectEx)(_In_opt_ IDebugHostContext* context, _In_ VARIANT* intrinsicData, _In_
IDebugHostType* type, _COM_Outptr_ IModelObject** object) PURE;
```

CreateNoValue

The CreateNoValue method creates a "no value" object, boxes it into an IModelObject, and returns it. The returned model object has a kind of ObjectNoValue.

A "no value" object has several semantic meanings:

- (Depending on language), it can be considered the semantic equivalent of void, null, or undefined
- Any property accessor's GetValue method which returns success and a resulting "no value" object is indicating that the particular property has no value for the given instance and should be treated as if the property did not exist for that particular instance.
- Data model methods which do not semantically have a return value use this as a sentinel to indicate such (as a method must return a valid IModelObject).

CreateErrorObject

The CreateErrorObject method creates an "error object". The data model does not have the notion of exceptions and exception flow. Failure comes out of a property/method in two ways:

- A single failing HRESULT with no extended error information. Either there is no more information which can be given for the error or the error itself is self-explanatory from the returned HRESULT.
- A single failing HRESULT coupled with extended error information. The extended error information is an error object returned in the output argument of the property/method.

CreateTypedObject

The CreateTypedObject method is the method which allows a client to create a representation of a native/language object in the address space of a debug target. If the type of the newly created object (as indicated by the objectType argument) happens to match one or more type signatures registered with the data model manager as either canonical visualizers or extensions, those matching data models will automatically be attached to the created instance object before it is returned to the caller.

CreateTypedObjectReference

The CreateTypedObjectReference method is semantically similar to the CreateTypedObject method excepting that it creates a reference to the underlying native/language construct. The created reference is an object which has a kind of ObjectTargetObjectReference. It is not a native reference as the underlying language might support (e.g.: a C++ & or &&). It is entirely possible to have a ObjectTargetObjectReference to a C++ reference. An object of kind

ObjectTargetObjectReference can be converted to the underlying value through use of the Dereference method on IModelObject. The reference can also be passed to the underlying host's expression evaluator in order to assign back to the value in a language appropriate way.

CreateSyntheticObject

The CreateSyntheticObject method creates an empty data model object -- a dictionary of key/value/metadata tuples and concepts. At the time of creation, there are no keys nor concepts on the object. It is a clean slate for the caller to utilize.

CreateDataModelObject

The CreateDataModelObject method is a simple helper wrapper to create objects which are data models -- that is objects which are going to be attached as parent models to other objects. All such objects must support the data model concept via IDataModelConcept. This method creates a new blank synthetic object with no explicit context and adds the inpassed IDataModelConcept as the newly created object's implementation of the data model concept. This can similarly be accomplished with calls to CreateSyntheticObject and SetConcept.

CreateIntrinsicObject

The CreateIntrinsicObject method is the method which boxes intrinsic values into IModelObject. The caller places the value in a COM VARIANT and calls this method. The data model manager returns an IModelObject representing the object. Note that this method is also used to box fundamental IUnknown based types: property accessors, methods, contexts, etc... In such cases, the objectKind method indicates what kind of IUnknown based construct the object represents and the punkVal field of the passed variant is the IUnknown derived type. The type must be statically castable to the appropriate model interface (e.g.: IModelPropertyAccessor, IModelMethod, IDebugHostContext, etc...) in process. The VARIANT types that are supported by this method are VT_UI1, VT_I1, VT_UI2, VT_I2, VT_UI4, VT_I4, VT_UI8, VT_I8, VT_R4, VT_R8, VT_BOOL, VT_BSTR, and VT_UNKNOWN (for a specialized set of IUnknown derived types as indicated by the enumeration ModelObjectKind.

CreateTypedIntrinsicObject

The CreateTypedintrinsicObject method is similar to the CreateIntrinsicObject method excepting that it allows a native/language type to be associated with the data and carried along with the boxed value. This allows the data model to represent constructs such as native enumeration types (which are simply VT_UI* or VT_I* values). Pointer types are also created with this method. A native pointer in the data model is a zero extended 64-bit quantity representing an offset into the virtual address space of the debug target. It is boxed inside a VT_UI8 and is created with this method and a type which indicates a native/language pointer.

CreateMetadataStore

The CreateMetadataStore method creates a key store -- a simplified container of key/value/metadata tuples -- which is used to hold metadata that can be associated with properties and a variety of other values. A metadata store may have a single parent (which in turn can have a single parent). If a given metadata key is not located in a given store, its parents are checked. Most metadata stores do not have parents. It does, however, provide a way of sharing common metadata easily.

CreateTypedIntrinsicObjectEx

The CreateTypedIntrinsicObjectEx method is semantically similar to the CreateTypedIntrinsicObject method. The only difference between the two is that this method allows the caller to specify the context in which the intrinsic data is valid. If no context is passed, the data is considered valid in whatever context is inherited from the type argument (how CreateTypedIntrinsicObject behaves). This allows for the creation of typed pointer values in the debug target which require more specific context than can be inherited from the type.

Extensibility / Registration Methods The following set of methods manages the extensibility mechanism of the data model, allowing a client to extend or register existing models or ask the data model to automatically attach a given parent model on native types which match a given criterion.

```
    STDMETHOD(GetModelForTypeSignature)(_In_ IDebugHostTypeSignature* typeSignature, _Out_ IModelObject**
dataModel) PURE;
    STDMETHOD(GetModelForType)(_In_ IDebugHostType* type, _COM_Outptr_ IModelObject** dataModel,
_COM_Outptr_opt_ IDebugHostTypeSignature** typeSignature, _COM_Outptr_opt_ IDebugHostSymbolEnumerator**
wildcardMatches) PURE;
    STDMETHOD(RegisterModelForTypeSignature)(_In_ IDebugHostTypeSignature* typeSignature, _In_ IModelObject*
dataModel) PURE;
    STDMETHOD(UnregisterModelForTypeSignature)(_In_ IModelObject* dataModel, _In_opt_ IDebugHostTypeSignature*
typeSignature) PURE;
    STDMETHOD(RegisterExtensionForTypeSignature)(_In_ IDebugHostTypeSignature* typeSignature, _In_
IModelObject* dataModel) PURE;
    STDMETHOD(UnregisterExtensionForTypeSignature)(_In_ IModelObject* dataModel, _In_opt_
IDebugHostTypeSignature* typeSignature) PURE;
    STDMETHOD(GetRootNamespace)(_COM_Outptr_ IModelObject** rootNamespace) PURE;
    STDMETHOD(RegisterNamedModel)(_In_ PCWSTR modelName, _In_ IModelObject *modeObject) PURE;
    STDMETHOD(UnregisterNamedModel)(_In_ PCWSTR modelName) PURE;
    STDMETHOD(AcquireNamedModel)(_In_ PCWSTR modelName, _COM_Outptr_ IModelObject **modelObject) PURE;
```

GetModelForTypeSignature

The GetModelForTypeSignature method returns the data model that was registered against a particular type signature via a prior call to the RegisterModelForTypeSignature method. The data model returned from this method is considered the canonical visualizer for any type which matches the passed type signature. As a canonical visualizer, that data model takes over the display of the type. Display engines will, by default, hide native/language constructs of the object in favor of the view of the object presented by the data model.

GetModelForType

The GetModelForType method returns the data model which is the canonical visualizer for a given type instance. In effect, this method finds the best matching type signature which was registered with a prior call to the RegisterModelForTypeSignature method and returns the associated data model.

RegisterModelForTypeSignature

The RegisterModelForTypeSignature method is the primary method that a caller utilizes to register a canonical visualizer for a given type (or set of types). A canonical visualizer is a data model which, in effect, takes over the display of a given type (or set of types). Instead of the native/language view of the type being displayed in any debugger user interface, the view of the type as presented by the registered data model is displayed (along with a means of getting back to the native/language view for a user who desires it).

UnregisterModelForTypeSignature

The UnregisterModelForTypeSignature method undoes a prior call to the RegisterModelForTypeSignature method. This method can either remove a given data model as the canonical visualizer for types matching a particular type signature or it can remove a given data model as the canonical visualizer for every type signature under which that data model is registered.

RegisterExtensionForTypeSignature

The RegisterExtensionForTypeSignature method is similar to the RegisterModelForTypeSignature method with one key difference. The data model which is passed to this method is not the canonical visualizer for any type and it will not take over the display of the native/language view of that type. The data model which is passed to this method will automatically be added as a parent to any concrete type which matches the supplied type signature. Unlike the RegisterModelForTypeSignature method, there is no limit on identical or ambiguous type signatures being registered as extensions to a given type (or set of types). Every extension whose type signature matches a given concrete type instance will cause the data model registered via this method to automatically be attached to newly created objects as parent models. This, in effect, allows an arbitrary number of clients to extend a type (or set of types) with new fields or functionality.

UnregisterExtensionForTypeSignature

The UnregisterExtensionForTypeSignature method undoes a prior call to RegisterExtensionForTypeSignature. It unregisters a particular data model as an extension for either a particular type signature or as an extension for all type signatures against which the data model was registered.

GetRootNamespace

The GetRootNamespace method returns the data model's root namespace. This is an object which the data model manages and into which the debug host places certain objects.

RegisterNamedModel

The RegisterNamedModel method registers a given data model under a well known name so that it can be found by clients wishing to extend it. This is the primary purpose of the API -- to publish a data model as something which can be extended by retrieving the model registered under this well known name and adding a parent model to it.

UnregisterNamedModel

The UnregisterNamedModel method undoes a prior call to RegisterNamedModel. It removes the association between a data model and a name under which it can be looked up.

AcquireNamedModel

A caller who wishes to extend a data model which is registered under a given name calls the AcquireNamedModel method in order to retrieve the object for the data model they wish to extend. This method will return whatever data model was registered via a prior call to the RegisterNamedModel method. As the primary purpose of the AcquireNamedModel method is to extend the model, this method has a special behavior if no model has been registered under the given name yet. If no model has been registered under the given name yet, a stub object is created, temporarily registered under the given name, and returned to the caller. When the real data model is registered via a call to the RegisterNamedModel method, any changes which were made to the stub object are, in effect, made to the real model. This removes many load order dependency issues from components which extend each other.

Helper Methods

The following methods are general helper methods which assist in performing complex operations on objects in the data model. While it is possible to perform these actions via other methods on the data model or its objects, these convenience methods make it significantly easier:

```
STDMETHOD(AcquireSubNamespace)(_In_ PCWSTR modelName, _In_ PCWSTR subNamespaceModelName, _In_ PCWSTR
accessName, _In_opt_ IKeyStore *metadata, _COM_Outptr_ IModelObject **namespaceModelObject) PURE;
```

AcquireSubNamespace

The AcquireSubNamespace method helps in the construction of something which might more traditionally look like a language namespace than a new object in a dynamic language. If, for instance, a caller wishes to categorize properties on a process object to make the process object more organized and the properties easier to discover, one method of doing this would be to create a sub-object for each category on the process object and placing those properties inside that object.

Related topics

This topic is part of a series which describes the interfaces accessible from C++, how to use them to build a C++ based debugger extension, and how to make use of other data model constructs (e.g.: JavaScript or NatVis) from a C++ data model extension.

Debugger Data Model C++ Overview

Debugger Data Model C++ Interfaces

Debugger Data Model C++ Additional Interfaces

Debugger Data Model C++ Concepts

Debugger Data Model C++ Scripting

Debugger Data Model C++ Additional Interfaces

10/23/2019 • 7 minutes to read • Edit Online

This topic describes some additional interfaces associated with the Debugger C++ Data Model, such as metadata, concepts and object enumeration.

Debugger Data Model Metadata Interfaces

One of the core notions in the data model is that an object (particularly a synthetic one) is a dictionary of key/value/metadata tuples. Each key can have an entire store of metadata associated with it that describes a variety of things surrounding the key and its potential value. Note that the metadata does not, in any way, change the value of the key. It is only ancillary information associated with the key and its value which may affect the presentation or other associated attributes of the key and its value.

In some senses, a metadata store is not all that different from the key/value/metadata tuples that are the essence of an object in the data model. It is, however, simplified from this view. A metadata store is represented by the IKeyStore interface. While also a collection of key/value/metadata tuples, there are limitations to what can be done with a metadata key store versus a model object:

- A key store can only have a single parent store -- it cannot have an arbitrary chain of parent models.
- A key store has no concepts. It can only have the dictionary of key/value/metadata tuples. This means that the keys present in a key store are static. They can not be created on demand by a dynamic language system.
- By convention only, the values in a metadata defined key store are restricted to basic values (intrinsics and property accessors)

While a key store can have an arbitrary number (and arbitrary naming) of keys, there are certain names that have defined semantic values. Presently, those names are:

KEY NAME	VALUE TYPE	DESCRIPTION
PreferredRadix	Integer: 2, 8, 10, or 16	Indicates what radix an ordinal value should be displayed in
PreferredFormat	Integer: as defined by the PreferredFormat enumeration	Indicates the preferred formatting type for display of the value
PreferredLength	Integer	For arrays and other containers, indicates how many elements should be displayed by default
FindDerivation	Boolean	Indicates whether the debug host should perform derived type analysis on the value before using (e.g.: displaying)
Help	String	Tool tip style help text for the key which can be presented by the user interface in an appropriately helpful way.

KEY NAME	VALUE TYPE	DESCRIPTION
ActionName	String	Indicates that the given method (one which takes no arguments and returns no values) is an action. The name of the action is specified in metadata. A user interface may utilize this name to present the option in a context menu or other appropriate interface
ActionIsDefault	Boolean	Only valid if the ActionName key is specified, indicates that this is the default action for the object.
ActionDescription	String	Only valid if the ActionName key is specified, this gives a tool tip style description for the action. Such text can be presented by the user interface in an appropriately helpful way.

Note that while keys in the metadata store can have their own metadata (ad infiniteum), there is currently no use for such. Most callers will specify null for any metadata parameters in methods on the IKeyStore interface.

The Core Metadata Interface: IKeyStore

The IKeyStore interface is defined as follows:

```
DECLARE_INTERFACE_(IKeyStore, IUnknown)
{
   STDMETHOD(GetKey)(_In_ PCWSTR key, _COM_Errorptr_opt_ IModelObject** object,
_COM_Outptr_opt_result_maybenull_ IKeyStore** metadata) PURE;
   STDMETHOD(SetKey)(_In_ PCWSTR key, _In_opt_ IModelObject* object, _In_opt_ IKeyStore* metadata) PURE;
   STDMETHOD(GetKeyValue)(_In_ PCWSTR key, _COM_Errorptr_opt_ IModelObject** object,
_COM_Outptr_opt_result_maybenull_ IKeyStore** metadata) PURE;
   STDMETHOD(SetKeyValue)(_In_ PCWSTR key, _In_ IModelObject* object) PURE;
   STDMETHOD(ClearKeys)() PURE;
}
```

GetKey

The GetKey method is analogous to the GetKey method on IModelObject. It will return the value of the specified key if it exists in the key store or the key store's parent store. Note that if the value of the key is a property accessor, the GetValue method will not be called on the property accessor. The actual IModelPropertyAccessor boxed into an IModelObject will be returned. It is typical that a client will call GetKeyValue for this reason.

SetKey

The SetKey method is analogous to the SetKey method on IModelObject. It is the only method which is capable of creating a key and associating metadata with it within the key store.

GetKeyValue

The GetKeyValue method is the first method a client will go to in order to find the value of a particular key within the metadata store. If the key specified by the key argument exists within the store (or it's parent store), the value of that key and any metadata associated with it will be returned. If the value of the key is a property accessor (an IModelPropertyAccessor boxed into an IModelObject), the GetValue method of the property accessor will automatically be called by GetKeyValue and the underlying value of the property returned.

SetKeyValue

The SetKeyValue method is analogous to the SetKeyValue method on IModelObject. This method is not capable of creating a new key within the metadata store. If there is an existing key as indicated by the key argument, its value will be set as indicated. If the key is a property accessor, the SetValue method will be called on the property accessor in order to set the underlying value. Note that metadata is typically static once created. Use of this method on a metadata key store should be infrequent.

ClearKeys

The ClearKeys method is analogous to the ClearKeys method on IModelObject. It will remove every key from the given metadata store. This method has no effect on any parent store.

Object Enumeration in the Data Model

Enumerating Objects in the Data Model

There are two core key enumeration interfaces in the data model: IKeyEnumerator and IRawEnumerator. While these are the two core interfaces, they can be used to enumerate objects in one of three styles:

Keys - The IKeyEnumerator interface can be acquired via a call to EnumerateKeys in order to enumerate the keys of an object and their values/metadata without resolving any underlying property accessors. This style of enumeration can return raw IModelPropertyAccessor values boxed into IModelObjects.

Values - The IKeyEnumerator and IRawEnumerator interfaces can be acquired via calls to either EnumerateKeyValues or EnumerateRawValues in order to enumerate the keys/raw values on an object and their values/metadata. Any property accessors present in the enumeration are automatically resolved via a call to the underlying GetValue method during such an enumeration.

References - The IKeyEnumerator and IRawEnumerator interfaces can be acquired via calls to either EnumerateKeyReferences or EnumerateRawReferences in order to enumerate references to the keys/raw values on an object. Such references can be saved and later used to get or set the underlying key or raw value.

KeyEnumerator: Enumeration of synthetic keys

The IKeyEnumerator interface is the single interface for the enumeration of all keys (by key, value, or reference) within an instance object and all the associated parent models in its parent model chain. The interface is defined as follows:

```
DECLARE_INTERFACE_(IKeyEnumerator, IUnknown)
{
    STDMETHOD(Reset)() PURE;
    STDMETHOD(GetNext)(_Out_ BSTR* key, _COM_Errorptr_opt_ IModelObject** value,
_COM_Outptr_opt_result_maybenull_ IKeyStore** metadata) PURE;
}
```

Reset

The Reset method resets the enumerator to the position it was at when it was first acquired (e.g.: before the first element in the enumeration). A subsequent call to GetNext will return the first enumerated key.

GetNext

The GetNext method both moves the enumerator forward and returns the key at that position in the enumeration.

IRawEnumerator: Enumeration of native or underlying language (C/C++) constructs

The IRawEnumerator interface is the single interface for the enumeration of all native/language constructs (by value or reference) within a object which represents a native construct within the address space of the debug target. The interface is defined as follows:

```
DECLARE_INTERFACE_(IRawEnumerator, IUnknown)
{
    STDMETHOD(Reset)() PURE;
    STDMETHOD(GetNext)(_Out_opt_ BSTR* name, _Out_opt_ SymbolKind *kind, _COM_Errorptr_opt_ IModelObject**
value) PURE;
}
```

Reset

The Reset method resets the enumerator to the position it was at when it was first acquired (e.g.: before the first element in the enumeration). A subsequent call to GetNext will return the first enumerated native/language construct.

GetNext

The GetNext method both moves the enumerator forward and returns the native/language construct at that position in the enumeration.

Related topics

This topic is part of a series which describes the interfaces accessible from C++, how to use them to build a C++ based debugger extension, and how to make use of other data model constructs (e.g.: JavaScript or NatVis) from a C++ data model extension.

Debugger Data Model C++ Overview

Debugger Data Model C++ Interfaces

Debugger Data Model C++ Objects

Debugger Data Model C++ Additional Interfaces

Debugger Data Model C++ Concepts

Debugger Data Model C++ Scripting

Debugger Data Model C++ Concepts

10/23/2019 • 17 minutes to read • Edit Online

This topic describes concepts in Debugger C++ Data Model .

Concepts in the Data Model

Synthetic objects in the data model are effectively two things:

- A dictionary of key/value/metadata tuples.
- A set of concepts (interfaces) that are supported by the data model. Concepts are interfaces that a client (as opposed to the data model) implements to provide a specified set of semantic behavior. The currently supported set of concepts are listed here.

CONCEPT INTERFACE	DESCRIPTION
IDataModelConcept	The concept is a parent model. If this model is automatically attached to a native type via a registered type signature, the InitializeObject method will automatically be called every time a new object of such type is instantiated.
IStringDisplayableConcept	The object can be converted to a string for display purposes.
IIterableConcept	The object is a container and can be iterated.
IIndexableConcept	The object is a container and can be indexed (accessed via random access) in one or more dimensions.
IPreferredRuntimeTypeConcept	The object understands more about types derived from it than the underlying type system is capable of providing and would like to handle its own conversions from static to runtime type.
IDynamicKeyProviderConcept	The object is a dynamic provider of keys and wishes to take over all key queries from the core data model. This interface is typically used as a bridge to dynamic languages such as JavaScript.
IDynamicConceptProviderConcept	The object is a dynamic provider of concepts and wishes to take over all concept queries from the core data model. This interface is typically used as a bridge to dynamic languages such as JavaScript.

The Data Model Concept: IDataModelConcept

Any model object which is attached to another model object as a parent model must directly support the data model concept. The data model concept requires support of an interface, IDataModelConcept defined as follows.

```
DECLARE_INTERFACE_(IDataModelConcept, IUnknown)
{
    STDMETHOD(InitializeObject)(_In_ IModelObject* modelObject, _In_opt_ IDebugHostTypeSignature*
matchingTypeSignature, _In_opt_ IDebugHostSymbolEnumerator* wildcardMatches) PURE;
    STDMETHOD(GetName)(_Out_ BSTR* modelName) PURE;
}
```

InitializeObject

A data model can be registered as the canonical visualizer or as an extension for a given native type through the data model manager's RegisterModelForTypeSignature or RegisterExtensionForTypeSignature methods. When a model is registered via either of these methods, the data model is automatically attached as a parent model to any native object whose type matches the signature passed in the registration. At the point where that attachment is automatically made, the InitializeObject method is called on the data model. It is passed the instance object, the type signature which caused the attachment, and an enumerator which produces the type instances (in linear order) which matched any wildcards in the type signature. The data model implementation may use this method call to initialize any caches it requires.

GetName

If a given data model is registered under a default name via the RegisterNamedModel method, the registered data model's IDataModelConcept interface must return that name from this method. Note that it is perfectly legitimate for a model to be registered under multiple names (the default or best one should be returned here). A model may be completely unnamed (so long as it is not registered under a name). In such circumstances, the GetName method should return E_NOTIMPL.

The String Displayable Concept: IStringDisplayableConcept

An object which wishes to provide a string conversion for display purposes can implement the string displayable concept through implementation of the IStringDisplayableConcept interface. The interface is defined as follows:

```
DECLARE_INTERFACE_(IStringDisplayableConcept, IUnknown)
{
    STDMETHOD(ToDisplayString)(_In_ IModelObject* contextObject, _In_opt_ IKeyStore* metadata, _Out_ BSTR*
displayString) PURE;
}
```

ToDisplayString

The ToDisplayString method is called whenever a client wishes to convert an object into a string to display (to console, in the UI, etc...). Such a string conversion should not be used for the basis of additional programmatic manipulation. The string conversion itself may be deeply influenced by the metadata passed to the call. A string conversion should make every attempt to honor the PreferredRadix and PreferredFormat keys.

The Iterable Concept: IIterableConcept and IModelIterator

An object which is a container of other objects and wishes to express the ability to iterate over those contained objects can support the iterable concept by an implementation of the IIterableConcept and IModelIterator interfaces. There is a very important relationship between support of the iterable concept and support of the indexable concept. An object which supports random access to the contained objects can support the indexable concept in addition to the iterable concept. In this case, the iterated elements must also produce a default index which, when passed to the indexable concept refer to the same object. A failure to satisfy this invariant will result in undefined behavior in the debug host.

The IIterableConcept is defined as follows:

```
DECLARE_INTERFACE_(IIterableConcept, IUnknown)
{
    STDMETHOD(GetDefaultIndexDimensionality)(_In_ IModelObject* contextObject, _Out_ ULONG64* dimensionality)
PURE;
    STDMETHOD(GetIterator)(_In_ IModelObject* contextObject, _Out_ IModelIterator** iterator) PURE;
}
```

The IModelIterator Concept is defined as follows:

```
DECLARE_INTERFACE_(IModelIterator, IUnknown)
{
   STDMETHOD(Reset)() PURE;
   STDMETHOD(GetNext)(_COM_Errorptr_ IModelObject** object, _In_ ULONG64 dimensions,
_Out_writes_opt_(dimensions) IModelObject** indexers, _COM_Outptr_opt_result_maybenull_ IKeyStore** metadata)
PURE;
}
```

IIterableConcept's GetDefaultIndexDimensionality

The GetDefaultIndexDimensionality method returns the number of dimensions to the default index. If an object is not indexable, this method should return 0 and succeed (S_OK). Any object which returns a non-zero value from this method is declaring support for a protocol contract which states:

- The object supports the indexable concept via support of IIndexableConcept
- The GetNext method of the IModelIterator returned from the GetIterator method of the iterable concept will return a unique default index for each produced element. Such index will have the number of dimensions as indicated here.
- Passing the indicies returned from the GetNext method of the IModelIterator to the GetAt method on the indexable concept (IIndexableConcept) will refer to the same object that GetNext produced. The same value is returned.

IIterableConcept's GetIterator

The GetIterator method on the iterable concept returns an iterator interface which can be used to iterate the object. The returned iterator must remember the context object that was passed to the GetIterator method. It will not be passed to methods on the iterator itself.

IModelIterator's Reset

The Reset method on an iterator returned from the iterable concept will restore the position of the iterator to where it was when the iterator was first created (before the first element). While it is strongly recommended that iterator's support the Reset method, it is not required. An iterator can be the equivalent of a C++ input iterator and only allow a single pass of forward iteration. In such case, the Reset method may fail with E_NOTIMPL.

IModelIterator's GetNext

The GetNext method moves the iterator forward and fetches the next iterated element. If the object is indexable in addition to being iterable and this is indicated by the GetDefaultIndexDimensionality argument returning a non-zero value, this method may optionally return the default indicies to get back to the produced value from the indexer. Note that a caller may choose to pass 0/nullptr and not retrieve any indicies. It is considered illegal for the caller to request partial indicies (e.g.: less than the number produced by GetDefaultIndexDimensionality).

If the iterator moved forward successfully but there was an error in reading the value of the iterated element, the method may return an error *AND* fill "object" with an error object. At the end of iteration of the contained elements, the iterator will return E_BOUNDS from the GetNext method. Any subsequent call (unless there has been an intervening Reset call) will also return E_BOUNDS.

The Indexable Concept: IIndexableConcept

An object which wishes to provide random access to a set of contents can support the indexable concept via support of the IIndexableConcept interface. Most objects which are indexable will be iterable as well through support of the iterable concept. This is not, however, required. If supported, there is an important relationship between the iterator and indexer. The iterator must support the GetDefaultIndexDimensionality, return a non-zero value from that method, and support the contract documented there. The indexer concept interface is defined as follows:

```
DECLARE_INTERFACE_(IIndexableConcept, IUnknown)
{
    STDMETHOD(GetDimensionality)(_In_ IModelObject* contextObject, _Out_ ULONG64* dimensionality) PURE;
    STDMETHOD(GetAt)(_In_ IModelObject* contextObject, _In_ ULONG64 indexerCount, _In_reads_(indexerCount)
IModelObject** indexers, _COM_Errorptr_ IModelObject** object, _COM_Outptr_opt_result_maybenull_ IKeyStore**
metadata) PURE;
    STDMETHOD(SetAt)(_In_ IModelObject* contextObject, _In_ ULONG64 indexerCount, _In_reads_(indexerCount)
IModelObject** indexers, _In_ IModelObject *value) PURE;
}
```

An example of using the indexer (and its interplay with the iterator) is shown below. This example iterates the contents of an indexable container and uses the indexer to get back to the value which was just returned. While that operation is functionally useless as written, it demonstrates how these interfaces interact. Note that the example below does not deal with memory allocation failure. It assumes a throwing new (which might be a poor assumption depending on the environment in which the code exists -- the COM methods of the data model cannot have C++ exceptions escape):

```
ComPtr<IModelObject> spObject;

//
// Assume we have gotten some object in spObject that is iterable (e.g.: an object which represents a
std::vector<SOMESTRUCT>)
//
ComPtr<IIterableConcept> spIterable;
ComPtr<IIndexableConcept> spIndexer;
if (SUCCEEDED(spObject->GetConcept(__uuidof(IIterableConcept), &spIterable, nullptr)) &&
    SUCCEEDED(spObject->GetConcept(__uuidof(IIndexableConcept), &spIndexable, nullptr)))
{
    ComPtr<IModelIterator> spIterator;

    //
    // Determine how many dimensions the default indexer is and allocate the requisite buffer.
    //
    ULONG64 dimensions;
    if (SUCCEEDED(spIterable->GetDefaultIndexDimensionality(spObject.Get(), &dimensions)) && dimensions > 0
&&
        SUCCEEDED(spIterable->GetIterator(spObject.Get(), &spIterator)))
    {
        std::unique_ptr<ComPtr<IModelObject>[]> spIndexers(new ComPtr<IModelObject>[dimensions]);

        //
        // We have an iterator.  Error codes have semantic meaning here.  E_BOUNDS indicates the end of
iteration.  E_ABORT indicates that
        // the debugger host or application is trying to abort whatever operation is occurring.  Anything
else indicates
        // some other error (e.g.: memory read failure) where the iterator MIGHT still produce values.
        //
        for(;;)
        {
            ComPtr<IModelObject> spContainedStruct;
            ComPtr<IKeyStore> spContainedMetadata;

            //
            // When we fetch the value from the iterator, it will pass back the default indicies.
```

```
            //
            HRESULT hr = spIterable->GetNext(&spContainedStruct, dimensions, reinterpret_cast<IModelObject
**>(spIndexers.get()), &spContainedMetadata);
            if (hr == E_BOUNDS || hr == E_ABORT)
            {
                break;
            }

            if (FAILED(hr))
            {
                //
                // Decide how to deal with failure to fetch an element.  Note that spContainedStruct *MAY*
contain an error object
                // which has detailed information about why the failure occurred (e.g.: failure to read
memory at address X).
                //
            }

            //
            // Use the indexer to get back to the same value.  We already have them, so there isn't much
functional point to this.  It simply
            // highlights the interplay between iterator and indexer.
            //
            ComPtr<IModelObject> spIndexedStruct;
            ComPtr<IKeyStore> spIndexedMetadata;

            if (SUCCEEDED(spIndexer->GetAt(spObject.Get(), dimensions, reinterpret_cast<IModelObject **>
(spIndexers.get()), &spIndexedStruct, &spIndexedMetadata)))
            {
                //
                // spContainedStruct and spIndexedStruct refer to the same object.  They may not have
interface equality.
                // spContainedMetadata and spIndexedMetadata refer to the same metadata store with the same
contents.  They may not have interface equality.
                //
            }
        }
    }
}
```

GetDimensionality

The GetDimensionality method returns the number of dimensions that the object is indexed in. Note that if the object is both iterable and indexable, the implementation of GetDefaultIndexDimensionality must agree with the implementation of GetDimensionality as to how many dimensions the indexer has.

GetAt

The GetAt method retrieves the value at a particular N-dimensional index from within the indexed object. An indexer of N-dimensions where N is the value returned from GetDimensionality must be supported. Note that an object may be indexable in different domains by different types (e.g.: indexable via both ordinals and strings). If the index is out of range (or could not be accessed), the method will return a failure; however, in such cases, the output object may still be set to an error object.

SetAt

The SetAt method attempts to set the value at a particular N-dimensional index from within the indexed object. An indexer of N-dimensions where N is the value returned from GetDimensionality must be supported. Note that an object may be indexable in different domains by different types (e.g.: indexable via both ordinals and strings). Some indexers are read-only. In such cases, E_NOTIMPL will be returned from any call to the SetAt method.

The Preferred Runtime Type Concept: IPreferredRuntimeTypeConcept

A debug host can be queried to make an attempt to determine the real runtime type of an object from a static type found in symbolic information. This conversion may be based on completely accurate information (e.g.: C++ RTTI) or may be based on strong heuristics such as the shape of any virtual function tables within the object. Some objects, however, cannot be converted from a static to a runtime type because they do not fit into the heuristics of the debug host (e.g.: they have no RTTI or virtual function tables). In such cases, a data model for an object can choose to override the default behavior and declare that it knows more about the "runtime type" of an object than the debug host is capable of understanding. This is done through the preferred runtime type concept and support of the IPreferredRuntimeTypeConcept interface.

The IPreferredRuntimeTypeConcept interface is declared as follows:

```
DECLARE_INTERFACE_(IPreferredRuntimeTypeConcept, IUnknown)
{
    STDMETHOD(CastToPreferredRuntimeType)(_In_ IModelObject* contextObject, _COM_Errorptr_ IModelObject**
object) PURE;
}
```

CastToPreferredRuntimeType

The CastToPreferredRuntimeType method is called whenever a client wishes to attempt to convert from a static type instance to the runtime type of that instance. If the object in question supports (through one of its attached parent models) the preferred runtime type concept, this method will be called to perform the conversion. This method may either return the original object (there is no conversion or it could not be analyzed), return a new instance of the runtime type, fail for non-semantic reasons (e.g.: out of memory), or return E_NOT_SET. The E_NOT_SET error code is a very special error code which indicates to the data model that the implementation does not want to override the default behavior and that the data model should fall back to whatever analysis is performed by the debug host (e.g.: RTTI analysis, examination of the shape of the virtual function tables, etc...)

The Dynamic Provider Concepts: IDynamicKeyProviderConcept and IDynamicConceptProviderConcept

While the data model itself normally handles key and concept management for objects, there are times where that notion is less than ideal. In particular, when a client wishes to create a bridge between the data model and something else which is truly dynamic (e.g.: JavaScript), it can be valuable to take over key and concept management from the implementation in the data model. As the core data model is the one and only implementation of IModelObject, this is instead done via a combination of two concepts: the dynamic key provider concept and the dynamic concept provider concept. While it would be typical to implement both or neither, there is no requirement for such.

If both are implemented, the dynamic key provider concept must be added before the dynamic concept provider concept. Both of these concepts are special. They effectively flip a switch on the object changing it from "statically managed" to "dynamically managed". These concepts can only be set if there are no keys/concepts managed by the data model on the object. Once these concepts are added to an object, the action of doing this is irrevocable.

There is an additional semantic difference around extensibility between an IModelObject which is a dynamic concept provider and one that is not. These concepts are intended to allow clients to create bridges between the data model and dynamic language systems such as JavaScript. The data model has a concept of extensibility that differs somewhat fundamentally from systems like JavaScript in that there is a tree of parent models rather than a linear chain like the JavaScript prototype chain. To allow a better relationship to such systems, an IModelObject which is a dynamic concept provider has a single data model parent. That single data model parent is a normal IModelObject which can have an arbitrary number of parent models as is typical for the data model. Any requests to the dynamic concept provider to add or remove parents are automatically redirected to the single parent. From an outsider's perspective, it looks as though the dynamic concept provider has a normal tree style chain of parent models. The implementer of the dynamic concept provider concept is the only object (outside of the core data model) that is aware of the intermediate single parent. That single parent can be linked against the dynamic

language system to provide a bridge (e.g.: placed into the JavaScript prototype chain).

The dynamic key provider concept is defined as follows:

```
DECLARE_INTERFACE_(IDynamicKeyProviderConcept, IUnknown)
{
    STDMETHOD(GetKey)(_In_ IModelObject *contextObject, _In_ PCWSTR key, _COM_Outptr_opt_result_maybenull_
IModelObject** keyValue, _COM_Outptr_opt_result_maybenull_ IKeyStore** metadata, _Out_opt_ bool *hasKey)
PURE;
    STDMETHOD(SetKey)(_In_ IModelObject *contextObject, _In_ PCWSTR key, _In_ IModelObject *keyValue, _In_
IKeyStore *metadata) PURE;
    STDMETHOD(EnumerateKeys)(_In_ IModelObject *contextObject, _COM_Outptr_ IKeyEnumerator **ppEnumerator)
PURE;
}
```

The dynamic concept provider concept is defined as follows:

```
DECLARE_INTERFACE_(IDynamicConceptProviderConcept, IUnknown)
{
    STDMETHOD(GetConcept)(_In_ IModelObject *contextObject, _In_ REFIID conceptId,
_COM_Outptr_result_maybenull_ IUnknown **conceptInterface, _COM_Outptr_opt_result_maybenull_ IKeyStore
**conceptMetadata, _Out_ bool *hasConcept) PURE;
    STDMETHOD(SetConcept)(_In_ IModelObject *contextObject, _In_ REFIID conceptId, _In_ IUnknown
*conceptInterface, _In_opt_ IKeyStore *conceptMetadata) PURE;
    STDMETHOD(NotifyParent)(_In_ IModelObject *parentModel) PURE;
    STDMETHOD(NotifyParentChange)(_In_ IModelObject *parentModel) PURE;
    STDMETHOD(NotifyDestruct)() PURE;
}
```

IDynamicKeyProviderConcept's GetKey

The GetKey method on a dynamic key provider is largely an override of the GetKey method on IModelObject. The dynamic key provider is expected to return the value of the key and any metadata associated with that key. In the event that the key is not present (but no other error occurs), the provider must return false in the hasKey parameter and succeed with S_OK. Failing this call is considered a failure to fetch a key and will explicitly halt the search for the key through the parent model chain. Returning false in hasKey and success will continue the search for the key. Note that it is perfectly legal for GetKey to return a boxed property accessor as the key. This would be semantically identical to the GetKey method on IModelObject returning a property accessor.

IDynamicKeyProviderConcept's SetKey

The SetKey method on a dynamic key provider is effectively an override of the SetKey method on IModelObject. This sets a key in the dynamic provider. It is effectively the creation of a new property on the provider. Note that a provider which does not support any notion of something like the creation of expando properties should return E_NOTIMPL here.

IDynamicKeyProviderConcept's EnumerateKeys

The EnumerateKeys method on a dynamic key provider is effectively an override of the EnumerateKeys method on IModelObject. This enumerates all the keys in the dynamic provider. The returned enumerator has several restrictions that must be honored by the implementation:

- It must behave as a call to EnumerateKeys and not EnumerateKeyValues or EnumerateKeyReferences. It must return the key values not resolving any underlying property accessors (if such concept exists in the provider).
- From the perspective of a single dynamic key provider, it is illegal to enumerate multiple keys of the same name that are physically distinct keys. This can happen on different providers that are attached through the parent model chain, but it cannot happen from the perspective of a single provider.

IDynamicConceptProviderConcept's GetConcept

The GetConcept method on a dynamic concept provider is effectively an override of the GetConcept method on IModelObject. The dynamic concept provider must return an interface for the queried concept if it exists as well as any metadata associated with that concept. If the concept does not exist on the provider, that must be indicated via a false value being returned in the hasConcept argument and a successful return. Failure of this method is a failure to fetch the concept and will explicitly halt the search for the concept. Returning false for hasConcept and a successful code will continue the search for the concept through the parent model tree.

IDynamicConceptProviderConcept's SetConcept

The SetConcept method on a dynamic concept provider is effectively an override of the SetConcept method on IModelObject. The dynamic provider will assign the concept. This may make the object iterable, indexable, string convertible, etc... Note that a provider which does not allow the creation of concepts on it should return E_NOPTIMPL here.

IDynamicConceptProviderConcept's NotifyParent

The NotifyParent call on a dynamic concept provider is used by the core data model to inform the dynamic provider of the single parent model which is created to allow for bridging the "multiple parent models" paradigm of the data model to more dynamic languages. Any manipulation of that single parent model will cause further notifications to the dynamic provider. Note that this callback is made immediately upon assignment of the dynamic concept provider concept.

IDynamicConceptProviderConcept's NotifyParentChange

The NotifyParent method on a dynamic concept provider is a callback made by the core data model when a static manipulation of the object's single parent model is made. For any given parent model added, this method will be called a first time when said parent model is added and a second time if/when said parent model is removed.

IDynamicConceptProviderConcept's NotifyDestruct

The NotifyDestruct method on a dynamic concept provider is a callback made by the core data model at the start of destruction of the object which is a dynamic concept provider. It provides additional clean up opportunities to clients which require it.

--

Related topics

This topic is part of a series which describes the interfaces accessible from C++, how to use them to build a C++ based debugger extension, and how to make use of other data model constructs (e.g.: JavaScript or NatVis) from a C++ data model extension.

Debugger Data Model C++ Overview

Debugger Data Model C++ Interfaces

Debugger Data Model C++ Objects

Debugger Data Model C++ Additional Interfaces

Debugger Data Model C++ Concepts

Debugger Data Model C++ Scripting

Debugger Data Model C++ Scripting

10/23/2019 • 36 minutes to read • Edit Online

This topic describes how to use Debugger Data Model C++ Debugger Data Model C++ scripting to support automation with the debugger engine using scripting.

Script Management in the Debugger Data Model

In addition to the Data Model Manager's role as the central authority on object creation and extensibility, it is also responsible for the management of an abstract concept of scripts. From the perspective of the Script Manager portion of the Data Model Manager, a script is something which can be dynamically loaded, unloaded, and potentially debugged by a provider in order to extend or provide new functionality to the data model.

A script provider is a component which bridges a language (e.g.: NatVis, JavaScript, etc...) to the data model. It registers one or more file extensions (e.g.: ".NatVis", ".js") which are handled by the provider allowing a debugger client or a user interface to allow for loading of script files with that particular extension by delegation to the provider.

The Core Script Manager: IDataModelScriptManager

The core script manager interface is defined as follows.

```
DECLARE_INTERFACE_(IDataModelScriptManager, IUnknown)
{
    STDMETHOD(GetDefaultNameBinder)(_COM_Outptr_ IDataModelNameBinder **ppNameBinder) PURE;
    STDMETHOD(RegisterScriptProvider)(_In_ IDataModelScriptProvider *provider) PURE;
    STDMETHOD(UnregisterScriptProvider)(_In_ IDataModelScriptProvider *provider) PURE;
    STDMETHOD(FindProviderForScriptType)(_In_ PCWSTR scriptType, _COM_Outptr_ IDataModelScriptProvider
**provider) PURE;
    STDMETHOD(FindProviderForScriptExtension)(_In_ PCWSTR scriptExtension, _COM_Outptr_
IDataModelScriptProvider **provider) PURE;
    STDMETHOD(EnumerateScriptProviders)(_COM_Outptr_ IDataModelScriptProviderEnumerator **enumerator) PURE;
}
```

GetDefaultNameBinder

The GetDefaultNameBinder method returns the data model's default script name binder. A name binder is a component which resolves a name within the context of an object. For instance, given the expression "foo.bar", a name binder is called upon to resolve the name bar in the context of object foo. The binder returned here follows a set of default rules for the data model. Script providers can use this binder to provide consistency in name resolution across providers.

RegisterScriptProvider

The RegisterScriptProvider method informs the data model that a new script provider exists which is capable of bridging a new language to the data model. When this method is called, the script manager will immediately call back the given script provider and inquire about the properties of the scripts it manages. If there is already a provider registered under the name or file extension which the given script provider indicates, this method will fail. Only a single script provider can be registered as the handler for a particular name or file extension.

UnregisterScriptProvider

The UnregisterScriptProvider method undoes a call to the RegisterScriptProvider method. The name and file extension given by the inpassed script provider will no longer be associated with it. It is important to note that

there may be a significant number of outstanding COM references to the script provider even after unregistration. This method only prevents the loading/creation of scripts of the type that the given script provider manages. If a script loaded by that provider is still loaded or has manipulated the object model of the debugger (or data model), those manipulations may still have references back into the script. There may be data models, methods, or objects which directly reference constructs in the script. A script provider must be prepared to deal with that.

FindProviderForScriptType

The FindProviderForScriptType method searches the script manager for a provider which has a script type string as indicated in this method. If one cannot be found, this method will fail; otherwise, such script provider will be returned to the caller.

EnumerateScriptProviders

The EnumerateScriptProviders method will return an enumerator which will enumerate every script provider that has been registered with the script manager via a prior call to the RegisterScriptProvider method.

Script Provider Enumeration: IDataModelScriptProviderEnumerator

The EnumerateScriptProviders method will return an enumerator of the following form:

```
DECLARE_INTERFACE_(IDataModelScriptProviderEnumerator, IUnknown)
{
    STDMETHOD(Reset)() PURE;
    STDMETHOD(GetNext)(_COM_Outptr_ IDataModelScriptProvider **provider) PURE;
}
```

Reset

The Reset method will move the enumerator to the position it was at prior to returning the first element.

GetNext

The GetNext method will move the enumerator forward one element and return the script provider which is at that element. When the enumerator hits the end of enumeration, E_BOUNDS will be returned. Calling the GetNext method after receiving this error will continue to return E_BOUNDS indefinitely.

Debugger Data Model C++ Host Interfaces for Scripting

The Host's Role in Scripting

The debug host exposes a series of very low level interfaces for understanding the nature of the type system of its debug target(s), evaluating expressions in the language of its debug target(s), etc... Normally, it is unconcerned with higher level constructs like scripting. Such is left to the overall debugger application or to extensions which provide these capabilities. There is, however, an exception to this. Any debug host which wants to participate in the overall scripting experience afforded by the data model needs to implement a few simple interfaces to provide contexts to scripts. In effect, the debug host is in control of where it wants the scripting environment to place functions and other script provided functionality within the namespace of the data model. Being involved in this process allows the host to allow (or not) the use of such functions in, for instance, its expression evaluator. The interfaces involved from the host's perspective here are:

INTERFACE	DESCRIPTION

INTERFACE	DESCRIPTION
IDebugHostScriptHost	The interface which indicates the capability of the debug host to take part in the scripting environment. This interface allows for the creation of contexts which inform scripting engines of where to place objects.
IDataModelScriptHostContext	A host interface which is used by the script provider as a container for the contents of the script. How the contents of a script surface other than the manipulations that it performs to the object model of the debugger application is up to the particular debug host. This interface allows the script provider to get information about where to place its contents. See Data Model C++ Scripting Interfaces later in this topic for more information.

The Debug Host's Script Host: IDebugHostScriptHost

The IDebugHostScriptHost interface is the interface used by a script provider to get a context from the debug host for a newly created script. This context includes an object (provided by the debug host) where the script provider can place any bridges between the data model and the scripting environment. Such bridges might, for instance, be data model methods which invoke script functions. Doing this allows a caller on the data model side to invoke script methods by utilization of the Call method on IModelMethod interface.

The IDebugHostScriptHost interface is defined as follows.

```
DECLARE_INTERFACE_(IDebugHostScriptHost, IUnknown)
{
    STDMETHOD(CreateContext)(_In_ IDataModelScript* script, _COM_Outptr_ IDataModelScriptHostContext**
scriptContext) PURE;
}
```

CreateContext

The CreateContext method is called by a script provider to create a new context in which to place the contents of the script. Such context is represented by the IDataModelScriptHostContext interface described in detail on the Data Model C++ Scripting Interfaces page.

Debugger Data Model C++ Scripting Interfaces

Scripting and Script Interfaces

The overall architecture of the data model allows a third party to define a bridge between some language and the object model of the data model. Typically, the language being bridged is a scripting language since the environment of the data model is very dynamic. A component which defines and implements this bridge between a language and the object model of the data model is called a script provider. When initialized, a script provider registers itself with the script manager portion of the data model manager and any interface which manages extensibility will subsequently allow for the editing, loading, unloading, and potentially debugging of scripts written to the language that the script provider manages.

Note that Debugging Tools for Windows presently defines two script providers.

- The NatVis Provider. This provider is embedded within DbgEng.dll and bridges between NatVis XML and data models, allowing visualization of native/language data types.
- The JavaScript Provider. This provider is contained within a legacy debugger extension: JsProvider.dll. It bridges between scripts written in the JavaScript language and the data model, allowing for arbitrary forms of debugger control and extensibility.

New providers can be written which bridge other languages (e.g.: Python, etc...) to the data model. Such would be presently encapsulated in legacy debugger extensions for loading purposes. The script provider itself should minimize the dependency with legacy engine interfaces and should only utilize the data model APIs where possible. This will allow the provider to be made portable to other environments with significantly greater ease.

There are two classes of interfaces related to script providers. The first class of interfaces is for general management of script providers and the scripts they manage. The second class of interfaces is for support of script debugging. While support for the first set is mandatory, support for the second is optional and may not make sense for every provider.

The general management interfaces are:

INTERFACE	DESCRIPTION
IDataModelScriptProvider	The core interface that a script provider must implement. This is the interface which is registered with the script manager portion of the data model manager in order to advertise the provider's support of a particular type of script and register against a particular file extension
IDataModelScript	An abstraction of a particular script which is being managed by the provider. Each script which is loaded or being edited has a separate IDataModelScript instance
IDataModelScriptClient	A client interface which is used by the script provider in order to communicate information to a user interface. Script providers do not implement this interface. The application hosting the data model which wishes to make use of script providers does. A script provider will call into methods of the script client to report status, errors, etc...
IDataModelScriptHostContext	A host interface which is used by the script provider as a container for the contents of the script. How the contents of a script surface other than the manipulations that it performs to the object model of the debugger application is up to the particular debug host. This interface allows the script provider to get information about where to place its contents.
IDataModelScriptTemplate	Script providers can provide one or more templates which serve as starting points for users to author scripts. A debugger application which provides a built-in editor can prefill new scripts with template content as advertised by the provider through this interface.
IDataModelScriptTemplateEnumerator	An enumerator interface that the script provider implements in order to advertise all the various templates it supports.
IDataModelNameBinder	A name binder -- an object which can associate a name in a context with a value. For a given expression such as "foo.bar", a name binder is able to bind the name "bar" in the context of object "foo" and produce a value or reference to it. Name binders are not typically implemented by a script provider; rather, the default binder can be acquired from the data model and used by the script provider

The debug interfaces are:

INTERFACE	DESCRIPTION
IDataModelScriptDebug	The core interface that a script provider must provide in order to make a script debuggable. The implementation class of the IDataModelScript interface must QueryInterface for IDataModelScriptDebug if the script is debuggable.
IDataModelScriptDebugClient	The user interface which wishes to provide the capability of script debugging implements the IDataModelScriptDebugClient interface. The script provider utilizes this interface to pass debug information back and forth (e.g.: events which occur, breakpoints, etc...)
IDataModelScriptDebugStack	The script provider implements this interface to expose the notion of a call stack to the script debugger.
IDataModelScriptDebugStackFrame	The script provider implements this interface to expose the notion of a particular stack frame within the call stack.
IDataModelScriptDebugVariableSetEnumerator	The script provider implements this interface to expose a set of variables. This set may represent the set of parameters to a function, the set of local variables, or the set of variables within a particular scope. The exact meaning is dependent upon how the interface was acquired.
IDataModelScriptDebugBreakpoint	The script provider implements this interface to expose the notion of and control of a particular breakpoint within the script.
IDataModelScriptDebugBreakpointEnumerator	The script provider implements this to enumerate all of the breakpoints which currently exist within the script (whether enabled or not).

The Core Script Provider: IDataModelScriptProvider

Any extension which wants to be a script provider must provide an implementation of the IDataModelScriptProvider interface and register such with the script manager portion of the data model manager via the RegisterScriptProvider method. This core interface which must be implemented is defined as follows.

```
DECLARE_INTERFACE_(IDataModelScriptProvider, IUnknown)
{
    STDMETHOD(GetName)(_Out_ BSTR *name) PURE;
    STDMETHOD(GetExtension)(_Out_ BSTR *extension) PURE;
    STDMETHOD(CreateScript)(_COM_Outptr_ IDataModelScript **script) PURE;
    STDMETHOD(GetDefaultTemplateContent)(_COM_Outptr_ IDataModelScriptTemplate **templateContent) PURE;
    STDMETHOD(EnumerateTemplates)(_COM_Outptr_ IDataModelScriptTemplateEnumerator **enumerator) PURE;
}
```

GetName

The GetName method returns the name of the type of (or language of) scripts which the provider manages as a string allocated via the SysAllocString method. The caller is responsible for freeing the returned string via SysFreeString. Examples of strings which might be returned from this method are "JavaScript" or "NatVis". The returned string is likely to appear in the user interface of the debugger application which is hosting the data model. No two script providers may return the same name (case insensitive).

GetExtension

The GetExtension method returns the file extension for scripts managed by this provider (without the dot) as a string allocated via the SysAllocString method. The debugger application hosting the data model (with scripting support) will delegate opening of script files with this extension to the script provider. The caller is responsible for freeing the returned string via SysFreeString. Examples of strings which might be returned from this method are "js" or "NatVis".

CreateScript

The CreateScript method is called to create a new script. The script provider must return a new and empty script represented by the returned IDataModelScript interface whenever this method is called. Note that this method is called regardless of whether a user interface is creating a new blank script for editing by the user or whether the debugger application is loading a script from disk. The provider does not get involved in file I/O. It merely handles the requests from the hosting application via streams passed to methods on IDataModelScript.

GetDefaultTemplateContent

The GetDefaultTemplateContent method returns an interface for the default template content of the provider. This is content that the script provider would like pre-populated in an edit window for a newly created script. If the script provider has no templates (or has no template content which is designated as the default content), the script provider may return E_NOTIMPL from this method.

EnumerateTemplates

The EnumerateTemplates method returns an enumerator which is capable of enumerating the variety of templates that are provided by the script provider. Template content is what the script provider wants to be "prefilled" into an edit window when creating a new script. If there are multiple different templates supported, those templates can be named (e.g.: "Imperative Script", "Extension Script") and the debugger application hosting the data model can choose how to present the "templates" to the user.

The Core Script Interface: IDataModelScript

The main interface which manages an individual script that is implemented by the provider is the IDataModelScript interface. A component implementing this interface is returned when the client wishes to create a new blank script and calls the CreateScript method on IDataModelScriptProvider.

Each script which is created by the provider should be in an independent silo. One script should not be able to impact another script except through explicit interaction with external objects via the data model. Two scripts, can for instance, both extend some type or concept (e.g.: the debugger's notion of what a process is). Either script can then access each other's fields via the external process object.

The interface is defined as follows.

```
DECLARE_INTERFACE_(IDataModelScript, IUnknown)
{
    STDMETHOD(GetName)(_Out_ BSTR *scriptName) PURE;
    STDMETHOD(Rename)(_In_ PCWSTR scriptName) PURE;
    STDMETHOD(Populate)(_In_ IStream *contentStream) PURE;
    STDMETHOD(Execute)(_In_ IDataModelScriptClient *client) PURE;
    STDMETHOD(Unlink)() PURE;
    STDMETHOD(IsInvocable)(_Out_ bool *isInvocable) PURE;
    STDMETHOD(InvokeMain)(_In_ IDataModelScriptClient *client) PURE;
}
```

GetName

The GetName method returns the name of the script as an allocated string via the SysAllocString function. If the script does not yet have a name, the method should return a null BSTR. It should not fail in this circumstance. If the script is explicitly renamed via a call to the Rename method, the GetName method should return the newly assigned name.

Rename

The Rename method assigns a new name to the script. It is the responsibility of the script implementation to save this name and return it upon any call to the GetName method. This is often called when a user interface chooses to Save As the script to a new name. Note that renaming the script may affect where the hosting application chooses to project the contents of the script.

Populate

The Populate method is called by the client in order to change or synchronize the "content" of the script. It is the notification that is made to the script provider that the code of the script has changed. It is important to note that this method does not cause execution of the script or changes to any of the objects that the script manipulates. This is merely a notification to the script provider that the content of the script has changed so that it may synchronize its own internal state.

Execute

The Execute method executes the content of the script as dictated by the last successful Populate call and modifies the object model of the debugger according to that content. If the language (or the script provider) defines a "main function" -- one that the author would want called upon clicking an imaginary "Execute Script" button in a user interface -- such "main function" is not called during an Execute operation. The Execute operation can be considered to perform initialization and object model manipulations only (e.g.: executing root code and setting up extensibility points).

Unlink

The Unlink method undoes the Execute operation. Any object model manipulations or extensibility points established during the execution of the script are undone. After an Unlink operation, the script may be re-executed via a call to Execute or it may be released.

IsInvocable

The IsInvocable method returns whether or not the script is invocable -- that is, whether it has a "main function" as defined by its language or provider. Such a "main function" is conceptually something that the script author would want called if an imaginary "Execute Script" button were pressed in a user interface.

InvokeMain

If the script has a "main function" which is intended to execute from a UI invocation, it indicates such via a true return from the IsInvocable method. The user interface can then call the InvokeMain method to actually "invoke" the script. Note that this is distinct from *Execute* which runs all root code and bridges the script to the namespace of the underlying host.

**The Script Client: IDataModelScriptClient **

An application hosting the data model that wants to manage scripts and have a user interface (whether graphical or console) around this notion implements the IDataModelScriptClient interface. This interface is passed to any script provider during execution or invocation or a script in order to pass error and event information back to the user interface.

The IDataModelScriptClient interface is defined as follows.

```
DECLARE_INTERFACE_(IDataModelScriptClient, IUnknown)
{
   STDMETHOD(ReportError)(_In_ ErrorClass errClass, _In_ HRESULT hrFail, _In_opt_ PCWSTR message, _In_ ULONG
line, _In_ ULONG position) PURE;
}
```

ReportError

If an error occurs during execution or invocation of the script, the script provider calls the ReportError method to notify the user interface of the error.

The Host Context for a Script: IDataModelScriptHostContext

The debug host has some influence over how and where it projects data model script content. It is expected that each script ask the host for a context in which to place bridges to the script (e.g.: function objects that can be called, etc...). This context is retrieved via calling the CreateContext method on IDebugHostScriptHost and getting an IDataModelScriptHostContext.

The IDataModelScriptHostContext interface is defined as follows.

```
DECLARE_INTERFACE_(IDataModelScriptHostContext, IUnknown)
{
   STDMETHOD(NotifyScriptChange)(_In_ IDataModelScript* script, _In_ ScriptChangeKind changeKind) PURE;
   STDMETHOD(GetNamespaceObject)(_COM_Outptr_ IModelObject** namespaceObject) PURE;
}
```

NotifyScriptChange

It is required that a script provider notify the debug host upon certain operations occurring with a method call to the NotifyScriptChange method on the associated context. Such operations are defined as members of the ScriptChangeKind enumeration

GetNamespaceObject

The GetNamespaceObject method returns an object into which the script provider can place any bridges between the data model and the script. It is here, for instance, that the script provider might place data model method objects (IModelMethod interfaces boxed into IModelObject) whose implementation calls into correspondingly named functions in the script.

Templates for Newly Created Scripts: IDataModelScriptTemplate

Script providers that want to present pre-filled content for new scripts (e.g.: to aid users writing scripts in a debugger user interface) can do so by providing one or more script templates. Such templates are components which implement the IDataModelScriptTemplate interface and are returned via either the GetDefaultTemplate method or EnumerateTemplates method on the script provider.

The IDataModelScriptTemplate interface is defined as follows.

```
DECLARE_INTERFACE_(IDataModelScriptTemplate, IUnknown)
{
   STDMETHOD(GetName)(_Out_ BSTR *templateName) PURE;
   STDMETHOD(GetDescription)(_Out_ BSTR *templateDescription) PURE;
   STDMETHOD(GetContent)(_COM_Outptr_ IStream **contentStream) PURE;
}
```

GetName

The GetName method returns a name of the template. This may fail with E_NOTIMPL if the template does not have a name. The single default template (if such exists) is not required to have a name. All other templates are. These names may be presented in a user interface as part of a menu to select which template is to be created.

GetDescription

The GetDescription method returns a description of the template. Such description would be presented to the user in more descriptive interfaces to help the user understand what the template is designed to do. The template

may return E_NOTIMPL from this method if it does not have a description.

GetContent

The GetContent method returns the content (or code) of the template. This is what would be pre-filled into the edit window if a user elected to create a new script from this template. The template is responsible for creating (and returning) a standard stream over the content that the client can pull.

Enumeration of a Provider's Template Content: IDataModelScriptTemplateEnumerator

A script provider can provide one or more templates which pre-fill content into newly created scripts in some user interface. If any of these templates are provided, the script provider must implement an enumerator over them which is returned upon a call to the EnumerateTemplates method.

Such enumerator is an implementation of the IDataModelScriptTemplateEnumerator interface and is defined as follows.

```
DECLARE_INTERFACE_(IDataModelScriptTemplateEnumerator, IUnknown)
{
   STDMETHOD(Reset)() PURE;
   STDMETHOD(GetNext)(_COM_Outptr_ IDataModelScriptTemplate **templateContent) PURE;
}
```

Reset

The Reset method resets the enumerator to the position it was at when it was first created -- before the first template produced.

GetNext

The GetNext method moves the enumerator to the next template and returns it. At the end of enumeration, the enumerator returns E_BOUNDS. Once the E_BOUNDS marker has been hit, the enumerator will continue to produce E_BOUNDS errors indefinitely until a Reset call is made.

Resolving the Meaning of Names: IDataModelNameBinder

The data model provides a standard way for script providers to determine the meaning of a given name in a given context (e.g.: determining what bar means for foo.bar) that will operate across a variety of script providers. This mechanism is known as a name binder and is represented by the IDataModelNameBinder interface. Such a binder encapsulates a set of rules about how the name resolves and how to deal with conflict resolution where a name is defined multiple times on an object. Part of these rules include things such as how a projected name (one added by a data model) resolves against a native name (one in the type system of the language being debugged).

In order to provide a degree of consistency across script providers, the data model's script manager provides a default name binder'. This default name binder can be acquired via a call to the GetDefaultNameBinder method on the IDataModelScriptManager interface. The name binder interface is defined as follows.

```
DECLARE_INTERFACE_(IDataModelNameBinder, IUnknown)
{
   STDMETHOD(BindValue)(_In_ IModelObject* contextObject, _In_ PCWSTR name, _COM_Errorptr_ IModelObject**
value, _COM_Outptr_opt_result_maybenull_ IKeyStore** metadata) PURE;
   STDMETHOD(BindReference)(_In_ IModelObject* contextObject, _In_ PCWSTR name, _COM_Errorptr_ IModelObject**
reference, _COM_Outptr_opt_result_maybenull_ IKeyStore** metadata) PURE;
   STDMETHOD(EnumerateValues)(_In_ IModelObject* contextObject, _COM_Outptr_ IKeyEnumerator** enumerator)
PURE;
   STDMETHOD(EnumerateReferences)(_In_ IModelObject* contextObject, _COM_Outptr_ IKeyEnumerator** enumerator)
PURE;
}
```

BindValue

The BindValue method performs the equivalent of contextObject.name on the given object according to a set of binding rules. The result of this binding is a value. As a value, the underlying script provider cannot use the value to perform assignment back to name.

BindReference

The BindReference method is similar to BindValue in that it also performs the equivalent of contextObject.name on the given object according to a set of binding rules. The result of the binding from this method is, however, a reference instead of a value. As a reference, the script provider can utilize the reference to perform assignment back to name.

EnumerateValues

The EnumerateValues method enumerates the set of names and values which will bind against the object according to the rules of the BindValue method. Unlike the EnumerateKeys, EnumerateValues, and similar methods on IModelObject which may return multiple names with the same value (for base classes, parent models, and the like), this enumerator will only return the specific set of names which will bind with BindValue and BindReference. Names will never be duplicated. Note that there is a significantly higher cost of enumerating an object via the name binder than calling the IModelObject methods.

EnumerateReferences

The EnumerateReferences method enumerates the set of names and references to them which will bind against the object according to the rules of the BindReference method. Unlike the EnumerateKeys, EnumerateValues, and similar methods on IModelObject which may return multiple names with the same value (for base classes, parent models, and the like), this enumerator will only return the specific set of names which will bind with BindValue and BindReference. Names will never be duplicated. Note that there is a significantly higher cost of enumerating an object via the name binder than calling the IModelObject methods.

Debugger Data Model C++ Script Debugging Interfaces

The infrastructure for script providers in the data model also provides a concept around debugging scripts. Any script that wishes to expose debugging capabilities to the debug host and the debugger application hosting the data model can do so by having debuggable scripts implement the IDataModelScriptDebug interface in addition to the IDataModelScript interface. The presence of this interface on the script indicates to the infrastructure that it is debuggable.

While the IDataModelScriptDebug interface is the starting point to get access to the debug capabilities of a script provider, it is joined by a set of other interfaces in providing overall debug capabilities.

The debug interfaces are:

INTERFACE	DESCRIPTION
IDataModelScriptDebug	The core interface that a script provider must provide in order to make a script debuggable. The implementation class of the IDataModelScript interface must QueryInterface for IDataModelScriptDebug if the script is debuggable.
IDataModelScriptDebugClient	The user interface which wishes to provide the capability of script debugging implements the IDataModelScriptDebugClient interface. The script provider utilizes this interface to pass debug information back and forth (e.g.: events which occur, breakpoints, etc...)

INTERFACE	DESCRIPTION
IDataModelScriptDebugStack	The script provider implements this interface to expose the notion of a call stack to the script debugger.
IDataModelScriptDebugStackFrame	The script provider implements this interface to expose the notion of a particular stack frame within the call stack.
IDataModelScriptDebugVariableSetEnumerator	The script provider implements this interface to expose a set of variables. This set may represent the set of parameters to a function, the set of local variables, or the set of variables within a particular scope. The exact meaning is dependent upon how the interface was acquired.
IDataModelScriptDebugBreakpoint	The script provider implements this interface to expose the notion of and control of a particular breakpoint within the script.
IDataModelScriptDebugBreakpointEnumerator	The script provider implements this to enumerate all of the breakpoints which currently exist within the script (whether enabled or not).

The general management interfaces are:

INTERFACE	DESCRIPTION
IDataModelScriptProvider	The core interface that a script provider must implement. This is the interface which is registered with the script manager portion of the data model manager in order to advertise the provider's support of a particular type of script and register against a particular file extension
IDataModelScript	An abstraction of a particular script which is being managed by the provider. Each script which is loaded or being edited has a separate IDataModelScript instance
IDataModelScriptClient	A client interface which is used by the script provider in order to communicate information to a user interface. Script providers do not implement this interface. The application hosting the data model which wishes to make use of script providers does. A script provider will call into methods of the script client to report status, errors, etc...
IDataModelScriptHostContext	A host interface which is used by the script provider as a container for the contents of the script. How the contents of a script surface other than the manipulations that it performs to the object model of the debugger application is up to the particular debug host. This interface allows the script provider to get information about where to place its contents.
IDataModelScriptTemplate	Script providers can provide one or more templates which serve as starting points for users to author scripts. A debugger application which provides a built-in editor can prefill new scripts with template content as advertised by the provider through this interface.
IDataModelScriptTemplateEnumerator	An enumerator interface that the script provider implements in order to advertise all the various templates it supports.

INTERFACE	DESCRIPTION
IDataModelNameBinder	A name binder -- an object which can associate a name in a context with a value. For a given expression such as "foo.bar", a name binder is able to bind the name "bar" in the context of object "foo" and produce a value or reference to it. Name binders are not typically implemented by a script provider; rather, the default binder can be acquired from the data model and used by the script provider.

Making Scripts Debuggable: IDataModelScriptDebug

Any script which is debuggable indicates this capability via the presence of the IDataModelScriptDebug interface on the same component which implements IDataModelScript. The query for this interface by the debug host or the debugger application hosting the data model is what indicates the presence of the debug capability.

The IDataModelScriptDebug interface is defined as follows.

```
DECLARE_INTERFACE_(IDataModelScriptDebug, IUnknown)
{
   STDMETHOD_(ScriptDebugState, GetDebugState)() PURE;
   STDMETHOD(GetCurrentPosition)(_Out_ ScriptDebugPosition *currentPosition, _Out_opt_ ScriptDebugPosition
*positionSpanEnd, _Out_opt_ BSTR *lineText) PURE;
   STDMETHOD(GetStack)(_COM_Outptr_ IDataModelScriptDebugStack **stack) PURE;
   STDMETHOD(SetBreakpoint)(_In_ ULONG linePosition, _In_ ULONG columnPosition, _COM_Outptr_
IDataModelScriptDebugBreakpoint **breakpoint) PURE;
   STDMETHOD(FindBreakpointById)(_In_ ULONG64 breakpointId, _COM_Outptr_ IDataModelScriptDebugBreakpoint
**breakpoint) PURE;
   STDMETHOD(EnumerateBreakpoints)(_COM_Outptr_ IDataModelScriptDebugBreakpointEnumerator **breakpointEnum)
PURE;
   STDMETHOD(GetEventFilter)(_In_ ScriptDebugEventFilter eventFilter, _Out_ bool *isBreakEnabled) PURE;
   STDMETHOD(SetEventFilter)(_In_ ScriptDebugEventFilter eventFilter, _In_ bool isBreakEnabled) PURE;
   STDMETHOD(StartDebugging)(_In_ IDataModelScriptDebugClient *debugClient) PURE;
   STDMETHOD(StopDebugging)(_In_ IDataModelScriptDebugClient *debugClient) PURE;
}
```

GetDebugState

The GetDebugState method returns the current state of the script (e.g.: whether it is executing or not). The state is defined by a value within the ScriptDebugState enumeration.

GetCurrentPosition

The GetCurrentPosition' method returns the current position within the script. This may only be called when the script is broken into the debugger where a call to GetScriptState would return ScriptDebugBreak. Any other call to this method is invalid and will fail.

GetStack

The GetStack method gets the current call stack at the break position. This method may only be called when the script is broken into the debugger.

SetBreakpoint

The SetBreakpoint method sets a breakpoint within the script. Note that the implementation is free to adjust the inpassed line and column positions to move forward to an appropriate code position. The actual line and column numbers where the breakpoint was placed can be retrieved by method calls on the returned IDataModelScriptDebugBreakpoint interface.

FindBreakpointById

Each breakpoint which is created within the script via the SetBreakpoint method is assigned a unique identifier (a 64-bit unsigned integer) by the implementation. The FindBreakpointById method is used to get an interface to the breakpoint from a given identifier.

EnumerateBreakpoints

The EnumerateBreakpoints method returns an enumerator capable of enumerating every breakpoint which is set within a particular script.

GetEventFilter

The GetEventFilter method returns whether "break on event" is enabled for a particular event. Events which can cause "break on event" are described by a member of the ScriptDebugEventFilter enumeration.

SetEventFilter

The SetEventFilter method changes the "break on event" behavior for a particular event as defined by a member of the ScriptDebugEventFilter enumeration. A full list of available events (and a description of this enumeration) can be found in the documentation for the GetEventFilter method.

StartDebugging

The StartDebugging method "turns on" the debugger for a particular script. The act of starting debugging does not actively cause any execution break or stepping. It merely makes the script debuggable and provides a set of interfaces for the client to communicate with the debugging interface.

StopDebugging

The StopDebugging method is called by a client that wants to stop debugging. This method call may be made at any point after StartDebugging was made successfully (e.g.: during a break, while the script is executing, etc...). The call immediately ceases all debugging activity and resets the state back to before StartDebugging was called.

The Debugging Interface: IDataModelScriptDebugClient

The debug host or debugger application which wishes to provide an interface around script debugging must provide an implementation of the IDataModelScriptDebugClient interface to the script debugger via the StartDebugging method on the debug interface for the script.

The IDataModelScriptDebugClient is the communication channel across which debug events are passed and control goes from the script execution engine to a debugger interface. It is defined as follows.

```
DECLARE_INTERFACE_(IDataModelScriptDebugClient, IUnknown)
{
   STDMETHOD(NotifyDebugEvent)(_In_ ScriptDebugEventInformation *pEventInfo, _In_ IDataModelScript *pScript,
_In_opt_ IModelObject *pEventDataObject, _Inout_ ScriptExecutionKind *resumeEventKind) PURE;
}
```

NotifyDebugEvent

Whenever any event occurs which breaks into the script debugger, the debug code itself makes a call to the interface via the NotifyDebugEvent method. This method is synchronous. No execution of the script will resume until the interface returns from the event. The definition of the script debugger is intended to be simple: there are absolutely no nested events requiring processing. A debug event is defined by a variant record known as a ScriptDebugEventInformation. Which fields in the event information are valid is largely defined by the DebugEvent member. It defines the kind of event which occurred as described by a member of the ScriptDebugEvent enumeration.

The Call Stack: IDataModelScriptDebugStack

When an event occurs which breaks into the script debugger, the debugging interface will want to retrieve the call

stack for the break location. This is done through the GetStack method. Such stack is expressed via the IDataModelScriptDebugStack which is defined as indicated below.

Note that the overall stack may span multiple scripts and/or multiple script providers. The call stack which is returned from a single call to the GetStack method on a particular script's debug interface should only return the segment of the call stack within the bounds of that script. It is entirely possible that a script debug engine can retrieve the call stack as spans multiple script contexts if two scripts of the same provider interact. The GetStack method should not return the portion of the stack which is in another script. Instead, if this situation can be detected, the stack frame which is the boundary frame into the script should mark itself as a transition frame via an implementation of the IsTransitionPoint and GetTransition methods on that stack frame. It is expected that the debugger interface will stitch together the overall stack from the multiple stack segments which exist.

It is imperative that transitions be implemented in this manner or the debug interface may direct inquiries about local variables, parameters, breakpoints, and other script specific constructs to the wrong script context! This will result in undefined behavior in the debugger interface.

```
DECLARE_INTERFACE_(IDataModelScriptDebugStack, IUnknown)
{
   STDMETHOD_(ULONG64, GetFrameCount)() PURE;
   STDMETHOD(GetStackFrame)(_In_ ULONG64 frameNumber, _COM_Outptr_ IDataModelScriptDebugStackFrame
**stackFrame) PURE;
}
```

GetFrameCount

The GetFrameCount method returns the number of stack frames in this segment of the call stack. If the provider can detect frames in different script contexts or of different providers, it should indicate this to the caller by implementation of the IsTransitionPoint and GetTransition methods on the entry frame into this stack segment.

GetStackFrame

The GetStackFrame gets a particular stack frame from the stack segment. The call stack has a zero based indexing system: the current stack frame where the break event occurred is frame 0. The caller of the current method is frame 1 (and so forth).

Examining State When Broken: IDataModelScriptDebugStackFrame

A particular frame of the call stack when broken into the script debugger can be retrieved via a call to the GetStackFrame method on the IDataModelScriptDebugStack interface representing the stack segment where the break occurred. The IDataModelScriptDebugStackFrame interface which is returned to represent this frame is defined as follows.

```
DECLARE_INTERFACE_(IDataModelScriptDebugStackFrame, IUnknown)
{
   STDMETHOD(GetName)(_Out_ BSTR *name) PURE;
   STDMETHOD(GetPosition)(_Out_ ScriptDebugPosition *position, _Out_opt_ ScriptDebugPosition
*positionSpanEnd, _Out_opt_ BSTR *lineText) PURE;
   STDMETHOD(IsTransitionPoint)(_Out_ bool *isTransitionPoint) PURE;
   STDMETHOD(GetTransition)(_COM_Outptr_ IDataModelScript **transitionScript, _Out_ bool
*isTransitionContiguous) PURE;
   STDMETHOD(Evaluate)(_In_ PCWSTR pwszExpression, _COM_Outptr_ IModelObject **ppResult) PURE;
   STDMETHOD(EnumerateLocals)(_COM_Outptr_ IDataModelScriptDebugVariableSetEnumerator **variablesEnum) PURE;
   STDMETHOD(EnumerateArguments)(_COM_Outptr_ IDataModelScriptDebugVariableSetEnumerator **variablesEnum)
PURE;
}
```

GetName

The GetName method returns the display name (e.g.: function name) of this frame. Such name will be displayed

within the stack backtrace presented to the user in the debugger interface.

GetPosition

The GetPosition method returns the position within the script represented by the stack frame. This method may only be called when the script is within a break represented by the stack in which this frame is contained. The line and column position within this frame is always returned. If the debugger is capable of returning the span of the "execution position" within the script, an ending position can be returned in the positionSpanEnd argument. If the debugger is not capable of this, the line and column values in the span end (if requested) should be set to zero.

IsTransitionPoint

The IDataModelScriptDebugStack interface represents a segment of a call stack -- that portion of the call stack which is contained within the context of one script. If the debugger is capable of detecting the transition from one script to another (or one script provider to another), it can indicate this by implementing the IsTransitionPoint method and returning true or false as appropriate. The call stack frame which entered the script where the segment applies should be considered a transition point. All other frames are not.

GetTransition

If a given stack frame is a transition point as determined by the IsTransition method (see the documentation there for a definition of transition points), the GetTransition method returns information about the transition. In particular, this method returns the previous script -- the one which made a call into the script represented by the stack segment containing this IDataModelScriptDebugStackFrame.

Evaluate

The Evaluate method evaluates an expression (of the language of the script provider) in the context of the stack frame represented by the IDataModelScriptDebugStackFrame interface on which this method was called. The result of the expression evaluation must be marshaled out of the script provider as an IModelObject. The properties and other constructs on the resulting IModelObject must all be able to be acquired while the debugger is in a break state.

EnumerateLocals

The EnumerateLocals method returns a variable set (represented by an IDataModelScriptDebugVariableSetEnumerator interface) for all local variables which are in scope in the context of the stack frame represented by the IDataModelScriptDebugStackFrame interface on which this method was called.

EnumerateArguments

The EnumerateArguments method returns a variable set (represented by an IDataModelScriptDebugVariableSetEnumerator interface) for all function arguments of the function called in the stack frame represented by the IDataModelScriptDebugStackFrame interface on which this method was called.

Looking at Variables: IDataModelScriptDebugVariableSetEnumerator

A set of variables in the script being debugged (whether those in a particular scope, the locals of a function, the arguments of a function, etc...) is represented by a variable set defined through the IDataModelScriptDebugVariableSetEnumerator interface:

```
DECLARE_INTERFACE_(IDataModelScriptDebugVariableSetEnumerator, IUnknown)
{
    STDMETHOD(Reset)() PURE;
    STDMETHOD(GetNext)(_Out_ BSTR *variableName, _COM_Outptr_opt_ IModelObject **variableValue,
_COM_Outptr_opt_result_maybenull_ IKeyStore **variableMetadata) PURE;
}
```

Reset

The Reset method resets the position of the enumerator to where it was immediately after creation -- that is, before the first element of the set.

GetNext

The GetNext method moves the enumerator to the next variable in the set and returns the variable's name, value, and any metadata associated with it. If the enumerator has hit the end of the set, the error E_BOUNDS is returned. Once the E_BOUNDS marker has been returned from the GetNext method, it will continue to produce E_BOUNDS when called again unless an intervening Reset call is made.

Breakpoints: IDataModelScriptDebugBreakpoint

Script breakpoints are set via the SetBreakpoint method on a given script's debug interface. Such breakpoints are represented both by a unique id and an implementation of the IDataModelScriptDebugBreakpoint interface which is defined as follows.

```
DECLARE_INTERFACE_(IDataModelScriptDebugBreakpoint, IUnknown)
{
    STDMETHOD_(ULONG64, GetId)() PURE;
    STDMETHOD_(bool, IsEnabled)() PURE;
    STDMETHOD_(void, Enable)() PURE;
    STDMETHOD_(void, Disable)() PURE;
    STDMETHOD_(void, Remove)() PURE;
    STDMETHOD(GetPosition)(_Out_ ScriptDebugPosition *position, _Out_opt_ ScriptDebugPosition
*positionSpanEnd, _Out_opt_ BSTR *lineText) PURE;
}
```

GetId

The GetId method returns the unique identifier assigned by the script provider's debug engine to the breakpoint. This identifier must be unique within the context of the containing script. The breakpoint identifier may be unique to the provider; however, that is not required.

IsEnabled

The IsEnabled method returns whether or not the breakpoint is enabled. A disabled breakpoint still exists and is still in the list of breakpoints for the script, it is merely "turned off" temporarily. All breakpoints should be created in the enabled state.

Enable

The Enable method enables the breakpoint. If the breakpoint was disabled, "hitting the breakpoint" after calling this method will cause a break into the debugger.

Disable

The Disable method disables the breakpoint. After this call, "hitting the breakpoint" after calling this method will not break into the debugger. The breakpoint, while still present, is considered "turned off".

Remove

The Remove method removes the breakpoint from its containing list. The breakpoint no longer semantically exists after this method returns. The IDataModelScriptDebugBreakpoint interface which represented the breakpoint is considered orphaned after the call. Nothing else can (legally) be done with it after this call other than releasing it.

GetPosition

The GetPosition method returns the position of the breakpoint within the script. The script debugger must return

the line and column within source code where the breakpoint is located. If it is capable of doing so, it can also return a span of source represented by the breakpoint by filling out an end position as defined by the positionSpanEnd argument. If the debugger is not capable of producing this span and the caller requests it, the Line and Column fields of the span's ending position should be filled in as zero indicating that the values cannot be provided.

Breakpoint Enumeration: IDataModelScriptDebugBreakpointEnumerator

If a script provider supports debugging, it must also keep track of all breakpoints associated with each and every script and be capable of enumerating those breakpoints to the debug interface. The enumerator for breakpoints is acquired via the EnumerateBreakpoints method on the debug interface for a given script and is defined as follows.

```
DECLARE_INTERFACE_(IDataModelScriptDebugBreakpointEnumerator, IUnknown)
{
   STDMETHOD(Reset)() PURE;
   STDMETHOD(GetNext)(_COM_Outptr_ IDataModelScriptDebugBreakpoint **breakpoint) PURE;
}
```

Reset

The Reset method resets the position of the enumerator to where it was just after the enumerator was created -- that is, before the first enumerated breakpoint.

GetNext

The GetNext method moves the enumerator forward to the next breakpoint to be enumerated and returns the IDataModelScriptDebugBreakpoint interface for that breakpoint. If the enumerator has reached the end of the enumeration, it returns E_BOUNDS. Once the E_BOUNDS error has been produced, subsequent calls to the GetNext method will continue to produce E_BOUNDS unless an intervening call to the Reset method has been made.

Related topics

This topic is part of a series which describes the interfaces accessible from C++, how to use them to build a C++ based debugger extension, and how to make use of other data model constructs (e.g.: JavaScript or NatVis) from a C++ data model extension.

Debugger Data Model C++ Overview

Debugger Data Model C++ Interfaces

Debugger Data Model C++ Objects

Debugger Data Model C++ Additional Interfaces

Debugger Data Model C++ Concepts

12/19/2018 • 2 minutes to read • Edit Online

IMPORTANT

This interface is under active development and will change.

The Code Namespace

Summary

The Code namespace contains attributes of code and disassembly. It enables creations of Disassembler objects that can disassemble given addresses or functions and provide detailed information about the assembly there and any variable or source information if availabe.

Sample

For an end-to-end example of how this namespace and objects and be used, check out the sample on GitHub - https://github.com/Microsoft/WinDbg-Samples/tree/master/CodeFlow

Object Methods

NAME	RETURN TYPE	SIGNATURE	DESCRIPTION
CreateDisassembler	disassembler	CreateDisassembler([architecture])	Creates a disassembler object of the specified architecture. Architecture may be one of "ARM", "ARM64", "X64", or "X86". If the architecture is not specified, X64 is assumed.
TraceDataFlow	collection of instructions	TraceDataFlow([address])	Looks at the instruction at the specified *address* (or the current instruction pointer if no address is specified) and all of its source operands. This method walks backwards through the control flow of the function looking for any instruction which influenced the source operands of the traced instruction. **This method requires loading the CodeFlow extension found here.**

Remarks

CreateDisassembler defaults to "X64" for the time being, at some point this behavior will change to pull the architecture of the module at the current thread's instruction pointer.

Disassembler Objects

12/19/2018 • 2 minutes to read • Edit Online

Summary

Disassembler objects enable the ability to disassemble code for a specific architecture.

Object Methods

NAME	RETURN TYPE	SIGNATURE	DESCRIPTION
DisassembleBlocks	collection of basic block	DisassembleBlocks(address)	Starts disassembling at *address* and returns a collection of basic blocks. The disassembly here is linearly forward from *address* on an instruction-by-instruction basis. Since this is not performing complete flow analysis of a function, it is entirely possible that there may be jumps into the middle of blocks returned by this method. There will only be a single exit point from each; however.
DisassembleInstructions	collection of instruction	DisassembleInstructions(address)	Starts disassembling at *address*.
DisassembleFunction	collection of basic block	DisassembleFunction(address)	Assuming a function starts at *address*, this performs a complete flow analysis of the function. The result is a collection of basic blocks with one entry point and one exit point.
GetRegister	register	GetRegister(regId)	Returns a register object from the given register id.

Remarks

The disassembler provided here has significantly better disassembly output if full symbolic information is present for the disassembled function (e.g.: it will utilize address and operand size to determine what field of a struct/union is being touched).

A given instance of a disassembler may cache a significant amount of data in order to provide a better experience.

Basic Block Objects

12/19/2018 • 2 minutes to read • Edit Online

Summary

Basic Blocks are regions of code with (typically) one entry point and one exit point. The disassembler's DisassembleBlocks and DisassembleFunction methods both return collections of basic blocks. The DisassembleBlocks method does a simple analysis for basic blocks and may result in blocks with multiple entry points. DisassembleFunction will perform a complete flow analysis of the function resulting in basic blocks with a single entry and single exit.

Object Properties

NAME	DESCRIPTION
StartAddress	The starting address of the basic block.
EndAddress	The ending address of the basic block. The block is defined by the half-open set [*StartAddress*, *EndAddress*).
Instructions	A collection of instruction objects in the basic block.
InboundControlFlows	This property is only present on basic blocks which are the result of full flow analysis (e.g.: *DisassembleFunction*). It is a collection of control flow objects which describe what other blocks have inbound control flow links to this one.
OutboundControlFlows	This property is only present on basic blocks which are the result of full flow analysis (e.g.: *DisassembleFunction*). It is a collection of control flow objects which describe the outbound control flow links from this block to other blocks in the function.

Instruction Objects

12/19/2018 • 2 minutes to read • Edit Online

Summary

Instruction Objects describe a single machine instruction and are returned via either an instruction based disassembly or as part of the contents of a basic block object.

Object Properties

NAME	DESCRIPTION
Address	The address of the machine instruction.
Attributes	An instruction attributes object which describes details about the instruction.
CodeBytes	An array of bytes representing the bytes which comprise the machine instruction.
Length	The number of bytes that the instruction takes in memory.
LiveVariables	A collection of live variable objects which describe the data which the compiler optimizer has emitted for variables at this particular location.
Operands	A collection of operand objects describing the operands of the instruction.
SourceInformation	A source information object which describes the relationship between the machine instruction and higher level source code.
SourceDataFlow	A collection of instruction objects within the function that comprise the data flow for source operands of the machine instruction. **This method requires loading the CodeFlow extension found here.**

Instruction Attributes Objects

12/19/2018 • 2 minutes to read • Edit Online

Summary

The *Attributes* property of an instruction object contains a description of some of the details of an instruction.

Object Properties

NAME	DESCRIPTION
IsBranch	Indicates whether the instruction is any sort of branch instruction.
IsConditional	Indicates whether the result of the instruction is conditional (e.g.: a conditional branch).
IsCall	Indicates whether the instruction is any sort of call instruction.
IsReturn	Indicates whether the instruction is any sort of return instruction.

Operand Objects

12/19/2018 • 2 minutes to read • Edit Online

Summary

Individual operands of a machine instruction are described by operand objects. The string conversion of an operand will be a portion of the string conversion of an entire instruction.

Object Properties

NAME	DESCRIPTION
Attributes	An operand attributes object describing certain aspects of the operand.
Registers	A collection of register objects which describe all registers that the operand utilizes.

Operand Attributes Objects

12/19/2018 • 2 minutes to read • Edit Online

Summary

An operand of a machine instruction is described by an operand attributes object.

Object Properties

NAME	DESCRIPTION
HasImmediate	Indicates whether the operand has an immediate value as part of the operand.
IsInput	Indicates whether the operand is a data source for the instruction (an input to whatever the instruction does).
IsOutput	Indicates whether the operand is a data destination for the instruction (an output of whatever the instruction does).
IsMemoryReference	Indicates whether the operand is a memory reference.
IsImmediate	Indicates whether the operand is an immediate value. Such an operand will also have *HasImmediate* set to true.
IsRegister	Indicates whether the operand is simply a register.

Register Objects

12/19/2018 • 2 minutes to read • Edit Online

Summary

The registers which are used by operands within machine instructions are described by register objects. The string conversion of a register object is the name of the register.

Object Properties

NAME	DESCRIPTION
Id	A unique identifier (within the domain of a specific architecture) for the register.

Remarks

Currently, you can only get the register's name through string conversion.

Live Variable Objects

12/19/2018 • 2 minutes to read • Edit Online

Summary

When operating with full symbolic information, each instruction contains a collection of information about the variables which are live at that particular instruction and where they are located.

Object Properties

NAME	DESCRIPTION
LocationKind	A string describing what kind of location the variable is stored in (e.g.: "Register", "RegisterRelative", etc...)
Offset	The offset from the location where the variable is stored. For a variable stored at [rbp + 8], for instance, the offset would be 8.
Register	A register object which describes the register where the variable is stored or relative to.
VariableName	The name of the variable which is being described.

Source Information Objects

12/19/2018 • 2 minutes to read • Edit Online

Summary

Machine instructions are related to source code via a source information object.

Object Properties

NAME	DESCRIPTION
FunctionName	The name of the function being described.
FunctionAddress	The base address of the function being described.
FunctionOffset	The offset into the function for the location being described.

Control Flow Objects

12/19/2018 • 2 minutes to read • Edit Online

Summary

For fully analyzed disassembly, each `basic block` contains a set of control flow objects in both the InboundControlFlows and OutboundControlFlows properties.

Object Properties

NAME	DESCRIPTION
LinkedBlock	The basic block object on the other side of the link. If this is an inbound control flow, this refers to the basic block which had the branch instruction. If this is an outbound control flow, this refers to the basic block which is the target of a branch instruction.
LinkKind	Indicates what kind of control flow resulted in a link between the two blocks (e.g.: "FallThrough" or "Branch").
SourceInstruction	The source of the control flow link. This is the branch instruction or the last instruction in a basic block.
TargetInstruction	The destination of the control flow link. This is the branch target or the instruction after the last instruction of a basic block with a fall through.

The Collections Namespace

12/19/2018 • 2 minutes to read • Edit Online

Summary

The Collections namespace provides methods that allow for the creation and manipulation of collections of objects (e.g.: iterables). The API extension extends the existing Collections namespace with the following properties and methods:

Object Methods

NAME	SIGNATURE	DESCRIPTION
CreateArray	CreateArray([value]...)	Creates an indexable and iterable array out of the set of arguments passed to the method. This method is useful for creating the starting point of a query or flattening

12/19/2018 • 2 minutes to read • Edit Online

IMPORTANT

This interface is under active development and will change.

The FileSystem Namespace

Summary

The FileSystem namespace provides the properties and methods for manipulating the file system. This can be used from JavaScript for reading or writing files that are needed to support your debugger extension.

Sample

For a simple end-to-end example of how to use this namespace and these objects, check out the sample on GitHub - https://github.com/Microsoft/WinDbg-Samples/tree/master/FileSystem

Object Methods

NAME	RETURN TYPE	SIGNATURE	DESCRIPTION
CreateFile	file	CreateFile(path, [disposition])	Creates a new file at the specified path and opens it for writing. *Disposition* may be one of "OpenExisting", "CreateNew", or "CreateAlways".
CreateTempFile	file	CreateTempFile()	Creates a new temporary file in the %TEMP% folder and opens it for writing.
CreateTextReader	text reader	CreateTextReader(file | path, [encoding])	Creates a text reader from the given file object or path which will read text of the specified encoding. Encoding may be one of "Ascii", "Utf8", or "Utf16". If not specified, "Ascii" is the default.
CreateTextWriter	text writer	CreateTextWriter(file | path, [encoding])	Creates a text writer from the given file object or path which will write text of the specified encoding. Encoding may be one of "Ascii", "Utf8", or "Utf16". If not specified, "Ascii" is the default.

NAME	RETURN TYPE	SIGNATURE	DESCRIPTION
DeleteFile		DeleteFile(path)	Deletes the file at the specified path.
FileExists	True or False	FileExists(path)	Returns true or false as to whether a file exists at the given path
OpenFile	file	OpenFile(path)	Opens a file at the specified path for reading.

Object Properties

NAME	DESCRIPTION
CurrentDirectory	A directory object representing the current working directory of the debugger process.
TempDirectory	A directory object representing the %TEMP% directory of the debugger process.

Directory Objects

12/19/2018 • 2 minutes to read • Edit Online

Summary

Directory objects represent and manipulate directories on the file system.

Object Methods

NAME	RETURN TYPE	SIGNATURE	DESCRIPTION
CreateFile	file	CreateFile(relativePath, [disposition])	Creates a file within the directory with the given disposition. Disposition may be one of "OpenExisting", "CreateNew", or "CreateAlways".
CreateSubDirectory	directory	CreateSubDirectory(name)	Creates a new subdirectory within the directory.
Delete		Delete()	Deletes the subdirectory if it is empty.
OpenFile	file	OpenFile(relativePath)	Opens an existing file for reading from the directory.

Object Properties

NAME	DESCRIPTION
Files	A collection of file within the directory.
SubDirectories	A collection of directory within the directory.

File Objects

1/3/2019 • 2 minutes to read • Edit Online

Summary

File objects are used to open, edit, and otherwise manipulate files on the file system.

Object Methods

NAME	RETURN TYPE	SIGNATURE	DESCRIPTION
Close		Close()	Closes the file.
Delete		Delete()	Deletes the file
Open		Open(disposition)	Opens the file with the given *disposition*. The *disposition* may be one of "OpenExisting", "CreateNew", or "CreateAlways".
ReadBytes	Array of bytes	ReadBytes(byteCount)	Reads the specified number of bytes from the file and returns an indexable and iterable array of those bytes.
WriteBytes		WriteBytes(bytes, [byteCount])	Writes the specified bytes to the file. If *byteCount* is not supplied, every byte which can be iterated out of *bytes* is written to the file; otherwise, the first *byteCount* bytes within *bytes* is written.

Object Properties

NAME	DESCRIPTION
Extension	A string containing the file extension.
Name	A string containing the name of the file.
Path	A string containing the fully qualified path to the file.
Position	The position of the current read/write cursor within the file.
Size	The size of the file in bytes.

Remarks

If the files are being used from a garbage collected environment such as a JavaScript script, they should be closed when no longer in use rather than waiting for a garbage collection to cause the file to close.

Text Reader Objects

12/19/2018 • 2 minutes to read • Edit Online

Summary

Text reader objects read text from a file.

Object Methods

NAME	SIGNATURE	DESCRIPTION
ReadLine	ReadLine()	Reads from the file until the next end-of-line (or the end-of-file marker) and returns the read as a string.
ReadLineContents	ReadLineContents()	Returns an iterable collection of strings. Each string in the collection represents one line in the file.

Text Writer Objects

12/19/2018 • 2 minutes to read • Edit Online

Summary

Text writer objects write text of the given encoding to a file.

Object Methods

NAME	SIGNATURE	DESCRIPTION
Write	Write(object)	Writes the string conversion of the given object to the file without a newline.
WriteLine	WriteLine(object)	Writes the string conversion of the given object to the file followed by a newline.
WriteContents	WriteContents(object)	Iterates *object* and writes the string conversion of each iterated element to the file without a newline between each.
WriteLineContents	WriteLineContents(object)	Iterates *object* and writes the string conversion of each iterated element to the file with a newline between each.

Glossary

9/5/2019 • 2 minutes to read • Edit Online

This glossary contains terms and acronyms related to the Microsoft Debugging Tools for Windows.

A

9/5/2019 • 2 minutes to read • Edit Online

accessible
A debugging session is *accessible* if the current target is suspended.

actual processor type
The type of the physical processor in the target computer.

See also effective processor type, executing processor type.

For more information, see Target Information.

arbitrary exception filter
An exception filter that has been manually added to the engine's list of event filters.

See also specific exception filter.

For more information, see Event Filters.

B

9/5/2019 • 2 minutes to read • Edit Online

blue screen
The blue character-mode screen displayed after a bug check occurs.

breakpoint
A location in the target or a target operation which will cause an event when triggered.

For more information, see Using Breakpoints.

breakpoint ID
The unique identifier for a breakpoint.

For more information, see Using Breakpoints.

breakpoint type
The method used to implement the breakpoint. There are two types of breakpoints: processor breakpoints and software breakpoints.

break status
A setting that influences how the debugger engine proceeds after an event. The break status indicates whether the event should break into the debugger, have a notification printed to the debugger console, or be ignored. The break status is part of an event filter.

See also *handling status*.

For more information, see the topics Controlling Exceptions and Events and Event Filters.

bug check
When Windows encounters hardware problems, inconsistencies within data necessary for its operation, or other severe errors, it shuts down and displays error information on a blue character-mode screen.

This shutdown is known variously as a bug check, kernel error, system crash, stop error, or, occasionally, trap. The screen display itself is referred to as a blue screen or stop screen. The most important information shown on the screen is a message code which gives information about the crash; this is known as a bug check code or stop code.

WinDbg or KD can configure the system so that a debugger is automatically contacted when such an error occurs. Alternatively, the system can be instructed to automatically reboot in case of such an error.

For more information, see Bug Checks (Blue Screens).

bug check code
The hexadecimal code indicating a specific type of bug check .

C

9/5/2019 • 2 minutes to read • Edit Online

C++ expression
An expression that can be evaluated by the C++ expression evaluator.

C call stack
See call stack.

call stack
The set of stack frames for each thread containing representing the function calls made by the thread. Each time a function call is made a new stack frame is pushed onto the top of the stack. When that function returns, the stack frame is popped off the stack.

Sometimes referred to as the or simply the .

callback object
See event callbacks, input callbacks, and output callbacks.

checked build
Two different builds of each NT-based operating system exist:

- The (or) of Windows is the end-user version of the operating system. For details, see free build.
- The (or) of Windows serves as a testing and debugging aid in the developing of the operating system and kernel-mode drivers. The checked build contains extra error checking, argument verification, and debugging information that is not available in the free build. , making it easier to trace the cause of problems in system software. A checked system or driver can help isolate and track down driver problems that can cause unpredictable behavior, result in memory leaks, or result in improper device configuration.

Although the checked build provides extra protection, it consumes more memory and disk space than the free build. System and driver performance is slower, because additional code paths are executed due to parameter checking and output of diagnostic messages, and some alternate implementations of kernel functions are used.

The checked build of Windows should not be confused with a driver that has been built in one of the Checked Build Environments of the Windows Driver Kit (WDK).

child symbol
A symbol that is contained in another symbol. For example, the symbol for a member in a structure is a child of the symbol for that structure.

client
See client object.

client object
A client object is used for interaction with the debugger engine. It holds per-client state, and provides implementations for the top-level interfaces in the debugger engine API.

client thread
The thread in which the client object was created. In general, a client's methods may be called only from this thread. The debugger engine uses this thread to make all calls to the callback object registered with the client.

code breakpoint
See software breakpoint.

crash dump file
A file that contains a snapshot of certain memory regions and other data related to an application or operating system. A crash dump file can be stored and then used to debug the application or operating system at a later time.

A user-mode crash dump file can be created by Windows when an application crashes, and a kernel-mode crash dump file can be created by special Windows routines when Windows itself crashes. There are several different types of crash dump files.

current process
The process that the debugger engine is currently controlling. When an event occurs, the current process is set to the event process.

current target
The target that the debugger engine is currently controlling. When an event occurs, the current target is set to the event target.

current thread
The thread that the debugger engine is currently controlling. When an event occurs, the current thread is set to the event thread.

D

9/5/2019 • 2 minutes to read • Edit Online

data breakpoint
See processor breakpoint.

DbgEng extension
A debugger extension based on the prototypes in the dbgeng.h header file. These extensions use the debugger engine API to communicate with the debugger engine.

debug build
See Checked Build.

debuggee
See target.

debugger
A debugger engine application that uses the full functionality of the debugger engine. For example, WinDbg, CDB, NTSD, and KD are debuggers.

debugger console
A fictional entity representing the source of the debugger engine input and the destination of its output. In reality, the debugger engine only uses input and output callbacks and has no notion of what is being used to implement them.

Typically, input is received from the Debugger Command window and output is sent to the same window. However, the input and output callbacks can provide many other sources of input and destinations for output, for example, remote connections, script files, and log files.

debugger engine
A library for manipulating debugging targets. Its interface is based on the prototypes in the dbgeng.h file. It is used by debuggers, extensions, and debugger engine applications.

debugger engine API
A powerful interface to control the behavior of the debugger engine. It may be used by debuggers, DbgEng extensions, and debugger engine applications. The prototypes for this interface are found in the dbgeng.h header file.

debugger engine application
A stand-alone application that uses the debugger engine API to control the debugger engine.

debugger extension
An external function that can run inside the debugger. Each extension is exported from a module known as an debugger extension DLL. The debugger engine invokes the debugger extension by calling its code within the DLL. Some debugger extensions ship with Debugging Tools for Windows. You can write your own extensions to automate any number of debugger features or to customize the output of the information accessible to the debugger.

Also referred to as an , or simply .

debugger extension DLL
A DLL containing debugger extensions. When the debugger engine loads the DLL these extensions become available for use within the debugger.

debugger extension library
See debugger extension DLL.

debugging client
An instance of the debugger engine acting as a proxy, sending debugger commands and I/O to the debugging server.

debugging server
An instance of the debugger engine acting as a host, listening for connections from debugging clients.

debugging session
The debugging session is the actual act of running a software debugging program, such as WinDbg, KD, or CDB, to debug a software component, system service, application, or operating system. The debugging session can also be run against a memory dump file for analysis.

A debugging session starts when a acquires a and lasts until all targets have been discarded.

default exception filter
The event filter which applies to exception events that do not match any other exception filters. The default exception filter is a specific exception filter.

dormant mode
A state in which a debugger program is running, but without a target or active session.

downstream store
A cache of symbols created by a symbol server. Typically this cache is on your local machine, while the symbol store is located remotely. If you have a chain of symbol servers, the downstream store can be located on any computer downstream from the symbol store.

dump file
See crash dump file.

dump target
A crash dump file that is being debugged.

E

10/23/2019 • 2 minutes to read • Edit Online

effective processor type
The processor type the debugger uses when dealing with the target computer. The effective processor type affects how the debugger sets breakpoints, accesses registers, gets stack traces, and performs other processor-specific actions.

engine
See debugger engine.

event
The debugger engine monitors some of the events that occur in its targets. These include a breakpoint being triggered, an exception, thread and process creation, module loading, and internal debugger engine changes.

event filter
A collection of rules that influence how the debugger engine proceeds after an event has occurred in a target. There are three types of event filters: specific event filters, specific exception filters, and arbitrary exception filters.

event callback objects
Instances of the IDebugEventCallbacks interface which have been registered with a client. The engine notifies the event callbacks whenever an event occurs.

event callbacks
See event callback objects.

event process
The process for which the last event occurred.

event target
The target for which the last event occurred.

event thread
The thread for which the last event occurred.

executing processor type
The type of the processor in the current processor context.

exception
An error condition resulting from the execution of a particular machine instruction or from the target indicating that it has encountered an error. Exceptions can be hardware or software-related errors.

An exception is an and is identified by its .

exception code
An identifier which determines the type of an exception event.

exception filter
An event filter for an exception event specified by exception code.

extension
See debugger extension.

extension command
See debugger extension.

F

9/5/2019 • 2 minutes to read • Edit Online

free build
Two different builds of each NT-based operating system exist:

- The (or) of Windows is the end-user version of the operating system. The system and drivers are built with full optimization, debugging asserts are disabled, and debugging information is stripped from the binaries. A free system and driver are smaller and faster, and it uses less memory.
- The (or) of Windows serves as a testing and debugging aid. For details, see checked build.

Distribution media containing the free build of the operating system do not have any special labels -- in other words, the CD containing the free build will just be labeled with the Windows version name, and no reference to the type of build.

first-chance exception
The first opportunity to handle an exception. If an exception is not handled by any handler on the first opportunity, the handlers are given a second chance.

H

9/5/2019 • 2 minutes to read • Edit Online

host
See host computer.

handling status
The handling status specifies whether the debugger engine should flag an exception event as handled or not. The handling status is part of an exception filter.

See also break status. For more information, see Controlling Exceptions and Events and Event Filters

host computer
The host computer is the computer that runs the debugging session. All debugger operations--such as executing commands and extensions, and symbol loading--are performed on the host computer.

In a typical two-system kernel debugging session, the debugger is running on the host computer, which is connected to the target computer being debugged.

10/23/2019 • 2 minutes to read • Edit Online

I/O Request Packet (IRP)
A data structure used to represent an I/O request and control its processing. An IRP structure consists of a header and one or more stack locations.

image
An executable, DLL, or driver that Windows has loaded as part of a user-mode process or the Windows kernel.

See also image file.

image file
The file from which an image was loaded.

implicit process
In kernel-mode debugging, the process used to determine which virtual address space to use when performing virtual to physical address translation. When an event occurs, the implicit process is set to the event process.

See also implicit thread.

For more information, see Threads and Processes.

implicit thread
In kernel-mode debugging, the thread used to determine some of the target's registers, including frame offset and instruction offset. When an event occurs, the implicit thread is set to the event thread.

inaccessible
A debugging session is *inaccessible* when all the targets are executing.

initial breakpoint
A breakpoint that automatically occurs near the beginning of a debugging session, after a reboot, or after a target application is restarted.

For more information, see Using Breakpoints.

input callback objects
Instances of the IDebugInputCallbacks interface which have been registered with a client. Whenever the debugger engine requires input it asks the input callbacks to provide it.

See also output callbacks.

For more information, see Using AMLI Debugger Commands.

input callbacks
See input callback objects.

interrupt
A condition that disrupts normal command execution and transfers control to an interrupt handler. I/O devices requiring service from the processor usually initiate interrupts.

Interrupt Request Level (IRQL)
The priority ranking of an interrupt. Each processor has an IRQL setting that threads can raise or lower. Interrupts that occur at or below the processor's IRQL setting are masked and will not interfere with the current operation. Interrupts that occur above the processor's IRQL setting take precedence over the current operation.

K

9/5/2019 • 2 minutes to read • Edit Online

KD connection server
A proxy used during some forms of kernel-mode remote debugging. It listens for connections from smart client and performs memory, processor, or Windows operations as requested by these remote clients.

See also debugging server.

For more information, see KD Connection Servers (Kernel Mode).

kernel
The kernel is the portion of the Windows operating system that manages and controls access to hardware resources. It performs thread scheduling and dispatching, interrupt and exception handling, and multiprocessor synchronization.

kernel error
See bug check.

kernel mode
Kernel-mode code has permission to access any part of the system, and is not restricted like user-mode code. It can gain access to any part of any other process running in either user mode or kernel mode.

Performance-sensitive operating system components run in kernel mode. In this way they can interact with the hardware and with each other without the overhead of context switch. All the kernel-mode components are fully protected from applications running in user mode. They can be grouped as follows:

- Executive.

  This contains the base operating system components such as memory management, process and thread management, security, I/O, interprocess communication.

- Kernel.

  This performs low-level functions such as thread scheduling, interrupt and exception dispatching, and multiprocessor synchronization. It also provides a set of routines and basic objects used by the Executive to implement higher-level semantics.

- Hardware Abstraction Layer (HAL).

  This handles all direct interface to hardware. It thus isolates the Windows Kernel, device drivers, and Windows Executive from platform-specific hardware differences.

- Window and Graphics Subsystem.

  This implements the graphical user interface (GUI) functions.

When a process erroneously accesses a portion of memory that is in use by another application or by the system, the lack of restrictions on kernel-mode processes forces Windows to stop the entire system. This is referred to as a bug check.

Malfunctioning hardware devices or device drivers, which reside in kernel mode, are often the culprits in bug checks.

kernel-mode target
See target computer.

kernel-mode debugging

A debugger session in which the target is running in kernel mode.

L

9/5/2019 • 2 minutes to read • Edit Online

live kernel-mode debugging
Kernel-mode debugging using live targets.

See also live user-mode debugging.

live target
A target application or target computer that is currently operational (as opposed to a dump target).

live user-mode debugging
User-mode debugging using live targets.

See also live user-mode debugging.

local context
See scope.

local debugging
This refers to a debugging session in which the debugger and the application to be debugged reside on the same computer.

M

9/5/2019 • 2 minutes to read • Edit Online

module

An image in a target process.

N

9/5/2019 • 2 minutes to read • Edit Online

nonpaged pool

A portion of system memory that will not be paged to disk.

O

10/23/2019 • 2 minutes to read • Edit Online

output callback objects
Instances of the IDebugOutputCallbacks interface which have been registered with a client object. All output from the debugger engine is sent to the output callbacks.

output callbacks
See output callback objects.

P

9/5/2019 • 2 minutes to read • Edit Online

page table
A process-specific table that maps virtual memory addresses to physical memory addresses.

Page Table Entry (PTE)
An item in the page table.

paged pool
A portion of system memory that can be paged to disk.

Note that this term does not only refer to memory that actually has actually been paged out to the disk - it includes any memory that the operating system is permitted to page.

paging
A virtual memory operation in which the memory manager transfers pages from memory to disk when physical memory becomes full. A *page fault* occurs when a thread accesses a page that is not in memory.

parent symbol
A *symbol* that is contains in other symbols, for example, a structure contains its member.

See also *child symbol*.

For more information, see Scopes and Symbol Groups.

primary client
A client object that has joined the current debugging session

For more information, see Client Objects.

process server
An instance of the debugger engine acting as a proxy, listening for connections from smart client and performing memory, processor, or Windows operations as requested by these remote clients.

See also *debugging server*.

For more information, see Process Servers (User Mode) and Process Server and Smart Client.

processor breakpoint
A breakpoint that is implemented by the processor. The debugger engine instructs the target's processor to insert this breakpoint.

See also software breakpoint. See also *software breakpoint*.

For more information, see Using Breakpoints.

R

9/5/2019 • 2 minutes to read • Edit Online

remote debugging
Remote debugging is the practice of using a remote connection to perform debugging.

Since user-mode debugging is usually done on one machine, remote user-mode debugging typically uses two machines. In this scenario, one computer contains the target application and a debugging server, while the other computer contains the debugging client. An alternate method is to have the target application and a process server on one computer, and a smart client on the other.

Since kernel-mode debugging is usually done on two machines, remote kernel-mode debugging requires three machines. The target computer is the computer being debugged. The server is attached to the target; it contains the kernel-mode debugger. The client is the computer which is controlling the session remotely. Typically, one computer is the being debugged, another contains the debugging server, and the third contains the debugging client.

In addition, there are other methods of remote debugging: using the Remote tool, using a KD connection server, or using a repeater. The method you should choose depends on the configuration of the machines in question and the available connections.

For more information, see Remote Debugging.

register
A very fast temporary memory location in the CPU.

register context
The full processor state which includes all the processor's registers.

For more information, see **Register Context**.

retail build
See free build.

S

9/5/2019 • 2 minutes to read • Edit Online

scope
The context that defines the local variables. The scope has three components: a stack frame, a current instruction, and a register context.

Sometimes referred to as *local context* or *lexical scope*.

second-chance exception
The second opportunity to handle an exception. This opportunity is only provided if the exception was not handled on the first chance.

smart client
An instance of the debugger engine acting as a host. The smart client is connected to a process server. or a KD connection server.

specific exception filter
An event filter for an exception for which the engine has a built-in filter. Most specific exception filters refer to specific types of exceptions (identified by exception code), but the default exception filter also qualifies as a specific exception filter.

specific event filter
An event filter for an event which is not an exception. The specific event filters are listed in **DEBUG_FILTER_XXX**.

specific filter
An event filter for an event for which the engine has a built-in filter.

software breakpoint
A breakpoint that is implemented by the debugger engine temporarily modifying the target's executable code. The breakpoint is triggered when the code is executed. The code modification is invisible to users of the debugger or the debugger engine API.

stack
See call stack.

stack frame
The memory in the call stack containing the data for a single function call. This includes the return address, parameters passed to the function, and local variables.

stop code
See bug check code.

stop error
See bug check.

stop screen
See blue screen.

subregister
A register that is contained within another register. When the subregister changes, the portion of the register that contains the subregister also changes.

suspended
A target, process, or thread is suspended if it has been blocked it from executing.

symbol
A unit of debugging information which provides interpretive information about the target in a debugging session. Examples of symbols include variables (local and global), functions, types and function entries. Information about symbols can include the name, type (if applicable), and the address or regisgter where it is stored, as well as any parent or child symbols. This information is stored in symbol files and is typically not available in the module itself.

The debugger engine can synthesize certain symbols when symbol files are not available (for example, exported symbols), but these symbols generally provide only minimal information.

symbol file
A supplemental file created when an application, library, driver, or operating system is built. A symbol file contains data which is not actually needed when running the binaries, but which is very useful in the debugging process.

symbol group
A group of symbols, typically representing all the local variables in a scope.

symbol type
Descriptive information about a symbol's representation, such as its layout in memory. This is part of the information a compiler uses to generate code to manipulate the symbol. It is also used by the debugger engine to manipulate symbols. The symbol type is distributed in symbol files.

synthetic module
A region of memory that the engine treats like a module. A synthetic module may contain synthetic symbols.

synthetic symbol
A memory address that the engine treats like a symbol.

system crash
See bug check.

T

9/5/2019 • 2 minutes to read • Edit Online

target
A target application, target computer, or dump target.

Sometimes referred to as the*debuggee* .

target application
An application that is being debugged in user-mode.

target computer
A computer that is being debugged in kernel-mode.

target ID
The unique identifier for a target.

target process
In user-mode debugging, an operating system process that is part of the target application.

In kernel-mode debugging, the virtual process representing the kernel.

target thread
In user-mode debugging, an operating system thread in the target application.

In kernel-mode debugging, a virtual thread representing a processor.

thread context
The state preserved by Windows when switching threads. This is similar to the register context, except that there is some extra kernel-only processor state that is part of the register context but not the thread context. This extra state is available as registers during kernel-mode debugging.

For more information, see Scopes and Symbol Groups.

transport layer
This controls communication between the host and target computers during remote debugging.

trap
See bug check.

type
See symbol type.

U

9/5/2019 • 2 minutes to read • Edit Online

user mode
Applications and subsystems run within Windows in *user mode*. Processes that run in user mode do so within their own virtual address spaces. They are restricted from gaining direct access to many parts of the system, including system hardware, memory that was not allocated for their use, and other portions of the system that might compromise system integrity. Because processes that run in user mode are effectively isolated from the system and other user-mode processes, they cannot interfere with these resources.

User-mode processes can be grouped in the following categories:

- System Processes

  These perform important functions and are integral part of the operating system. System processes include such items as the logon process and the session manager process.

- Server Processes

  These are operating system services such as the Event Log and the Scheduler. They can be configured to start automatically at boot time.

- Environment Subsystems

  These are used to create a complete operating system environment for the applications. Windows provides the following three environments: Win32, POSIX, and OS/2.

- User Applications

  There are five types: Win32, Windows 3.1, Microsoft MS-DOS, POSIX, and OS/2.

user-mode debugging
A debugger session in which the target is running in user mode.

user-mode target
See target application.

V

9/5/2019 • 2 minutes to read • Edit Online

virtual process

In kernel-mode debugging the debugger engine creates a single *virtual process* to represent the target's kernel.

virtual thread

In kernel-mode debugging the debugger engine creates a *virtual thread* for each processor in the target computer.

W

9/5/2019 • 2 minutes to read • Edit Online

WdbgExts API
An interface exposed by the debugger engine for extensions. It contains a subset of the functionality of the debugger engine API and can only be used by extensions.

WdbgExts extension
An extension based on the prototypes in the wdbgexts.h header file. These extensions use the WdbgExts API.

WOW64
An emulation layer in 64-bit Windows that can run 32-bit Windows applications. When debugging a 32-bit application running on 64-bit Windows, it is important to distinguish between the application itself and the WOW64 subsystem.

Debugger Operation

12/5/2018 • 2 minutes to read • Edit Online

In this section:

- Debugging Using Visual Studio
- Debugging Using WinDbg
- Debugging Using KD and NTKD
- Debugging Using CDB and NTSD
- Controlling the Target
- Enabling Postmortem Debugging
- Using the Debugger Command Window
- Using the WinDbg Graphical Interface
- Using Debugger Extensions
- Performing Remote Debugging

Debugging Using WinDbg

12/5/2018 • 2 minutes to read • Edit Online

This section describes how to perform basic debugging tasks using the WinDbg debugger.

Details are given in the following topics:

- Debugging a User-Mode Process Using WinDbg
- Debugging a UWP app using WinDbg
- Opening a Dump File Using WinDbg
- Live Kernel-Mode Debugging Using WinDbg
- Ending a Debugging Session in WinDbg
- Setting Symbol and Executable Image Paths in WinDbg
- Remote Debugging Using WinDbg
- Entering Debugger Commands in WinDbg
- Using the Command Browser Window in WinDbg
- Setting Breakpoints in WinDbg
- Viewing the Call Stack in WinDbg
- Assembly Code Debugging in WinDbg
- Source Code Debugging in WinDbg
- Viewing and Editing Memory in WinDbg
- Viewing and Editing Global Variables in WinDbg
- Viewing and Editing Local Variables in WinDbg
- Viewing and Editing Registers in WinDbg
- Controlling Processes and Threads in WinDbg
- Configuring Exceptions and Events in WinDbg
- Keeping a Log File in WinDbg

Debugging a User-Mode Process Using WinDbg

12/5/2018 • 5 minutes to read • Edit Online

You can use WinDbg to attach to a running process or to spawn and attach to a new process.

Attaching to a Running Process

There are several ways you can use WinDbg to attach to a running process. Regardless of the method you choose, you will need the process ID or the process name. The process ID is a number assigned by the operating system. For more information about how to determine the process ID and the process name, see Finding the Process ID.

WinDbg Menu

When WinDbg is in dormant mode, you can attach to a running process by choosing **Attach to a Process** from the **File** menu or by pressing **F6**.

In the **Attach to Process** dialog box, select the process you want to debug, and click **OK**.

Command Prompt

In a Command Prompt window, you can attach to a running process when you launch WinDbg. Use one of the following commands:

- **windbg -p** *ProcessID*
- **windbg -pn** *ProcessName*

where *ProcessID* is the Process ID of a running process or *ProcessName* is the name of a running process.

For more information about the command-line syntax, see **WinDbg Command-Line Options**.

Debugger Command Window

If WinDbg is already debugging one or more processes, you can attach to a running process by using the **.attach (Attach to Process)** command in the Debugger Command window.

The debugger always starts multiple target processes simultaneously, unless some of their threads are frozen or suspended.

If the **.attach** command is successful, the debugger attaches to the specified process the next time the debugger issues an execution command. If you use this command several times in a row, execution has to be requested by the debugger as many times as you use this command.

Attaching to a Running Process Noninvasively

If you want to debug a running process and interfere only minimally in its execution, you should debug the process noninvasively.

WinDbg Menu

When WinDbg is in dormant mode, you can noninvasively debug a running process by choosing **Attach to a Process** from the **File** menu or by pressing **F6**.

When the **Attach to Process** dialog box appears, select the **Noninvasive** check box. Then, select the line that contains the process ID and name that you want. (You can also enter the process ID in the **Process ID** box.) Finally, click **OK**.

Command Prompt

In a Command Prompt window, you can attach to a running process noninvasively when you launch WinDbg. Use one of the following commands:

windbg -pv -p *ProcessID* **windbg -pv -pn** *ProcessName* There are several other useful command-line options. For more information about the command-line syntax, see **WinDbg Command-Line Options**.

Debugger Command Window

If the debugger is already active, you can noninvasively debug a running process by using the **.attach -v (Attach to Process)** command in the Debugger Command window.

You can use the **.attach** command if the debugger is already debugging one or more processes invasively. You cannot use this command if WinDbg is dormant.

If the **.attach -v** command is successful, the debugger debugs the specified process the next time that the debugger issues an execution command. Because execution is not permitted during noninvasive debugging, the debugger cannot noninvasively debug more than one process at a time. This restriction also means that using the **.attach -v** command might make an existing invasive debugging session less useful.

Spawning a New Process

WinDbg can start a user-mode application and then debug the application. The application is specified by name. The debugger can also automatically attach to child processes (additional processes that the original target process started).

Processes that the debugger creates (also known as spawned processes) behave slightly differently than processes that the debugger does not create.

Instead of using the standard heap API, processes that the debugger creates use a special debug heap. You can force a spawned process to use the standard heap instead of the debug heap by using the _NO_DEBUG_HEAP environment variable or the **-hd** command-line option.

Also, because the target application is a child process of the debugger, it inherits the debugger's permissions. This permission might enable the target application to perform certain actions that it could not perform otherwise. For example, the target application might be able to affect protected processes.

WinDbg Menu

When WinDbg is in dormant mode, you can spawn a new process by choosing **Open Executable** from the **File** menu or by pressing CTRL+E.

When the Open Executable dialog box appears, enter the full path of the executable file in the **File name** box, or use the **Look in** list to select the path and file name that you want.

If you want to use any command-line parameters with the user-mode application, enter them in the **Arguments** box. If you want to change the starting directory from the default directory, enter the directory path in the **Start** directory box. If you want WinDbg to attach to child processes, select the **Debug child processes also** check box.

After you make your selections, click **Open**.

Command Prompt

In a Command Prompt window, you can spawn a new process when you launch WinDbg. Use the following command:

windbg [-o] *ProgramName* **[***Arguments***]**

The **-o** option causes the debugger to attach to child processes. There are several other useful command-line options. For more information about the command-line syntax, see **WinDbg Command-Line Options**.

Debugger Command Window

If WinDbg is already debugging one or more processes, you can create a new process by using the **.create (Create Process)** command in the Debugger Command window.

The debugger will always start multiple target processes simultaneously, unless some of their threads are frozen or suspended.

If the **.create** command is successful, the debugger creates the specified process the next time that the debugger issues an execution command. If you use this command several times in a row, execution has to be requested by the debugger as many times as you use this command.

You can control the application's starting directory by using the **.createdir (Set Created Process Directory)** command before **.create**. You can use the **.createdir -I** command or the **-noinh** command-line option to control whether the target application inherits the debugger's handles.

You can activate or deactivate the debugging of child processes by using the **.childdbg (Debug Child Processes)** command.

Reattaching to a Process

If the debugger stops responding or freezes, you can attach a new debugger to the target process. For more information about how to attach a debugger in this situation, see Reattaching to the Target Application.

Debugging a UWP app using WinDbg

6/25/2019 • 8 minutes to read • Edit Online

You can debug Universal Windows Platform (UWP) app using WinDbg. This approach would typically be used for advanced scenarios, where it is not possible to complete the debugging task using the built in Visual Studio debugger. For more information about debugging in Visual Studio, see Debugging in Visual Studio.

Attaching to a UWP app

Attaching to UWP process is the same as attaching to a user mode process. For example, in WinDbg you can attach to a running process by choosing **Attach to a Process from the File** menu or by pressing F6. For more information, see Debugging a User-Mode Process Using WinDbg.

A UWP app will not be suspended in the same ways that it does when not being debugged. To explicitly suspend/resume a UWP app, you can use the .suspendpackage and .resumepackage commands (details below). For general information on Process Lifecycle Management (PLM) used by UWP apps, see App lifecycle and Launching, resuming, and background tasks.

Launching and debugging a UWP app

The -plmPackage and -plmApp command line parameters instruct the debugger to launch an app under the debugger.

```
windbg.exe -plmPackage <PLMPackageName> -plmApp <ApplicationId> [<parameters>]
```

Since multiple apps can be contained within a single package, both <PLMPackage> and <ApplicationId> parameters are required. This is a summary of the parameters.

Parameter	Description
<PLMPackageName>	The name of the application package. Use the .querypackages command to lists all UWP applications. Do not provide a path to the location of the package, provide just the package name.
<ApplicationId>	The ApplicationId is located in the application manifest file and can be viewed using the .querypackage or .querypackages command as discussed in this topic. For more information about the application manifest file, see App package manifest.
[<parameters>]	Optional parameters passed to the App. Not all apps use or require parameters.

HelloWorld Sample

To demonstrate UWP debugging, this topic uses the HelloWorld example described in Create a "Hello, world" app (XAML).

To create a workable test app, it is only necessary to complete up to step three of the lab.

Locating the Full Package Name and AppId

Use the .querypackages command to locate the full package name and the AppId. Type .querypackages and then user CRTL+F to search up in the output for the application name, such as HelloWorld. When the entry is located using CTRL+F, it will show the package full name, for example *e24caf14-8483-4743-b80c-ca46c28c75df_1.0.0.0_x86__97ghe447vaan8* and the AppId of *App*.

Example:

```
0:000>  .querypackages
...
Package Full Name: e24caf14-8483-4743-b80c-ca46c28c75df_1.0.0.0_x86__97ghe447vaan8
Package Display Name: HelloWorld
Version: 1.0.0.0
Processor Architecture: x86
Publisher: CN=user1
Publisher Display Name: user1
Install Folder: c:\users\user1\documents\visual studio
2015\Projects\HelloWorld\HelloWorld\bin\x86\Release\AppX
Package State: Unknown
AppId: App
...
```

Viewing the Base Package Name in the in the Manifest

For troubleshooting, you may want to view the base package name in Visual Studio.

To locate the base package name in Visual Studio, click on the ApplicationManifest.xml file in project explorer. The base package name will be displayed under the packaging tab as "Package name". By default, the package name will be a GUID, for example *e24caf14-8483-4743-b80c-ca46c28c75df*.

To use notepad to locate the base package name, open the ApplicationManifest.xml file and locate the **Identity Name** tag.

```
  <Identity
    Name="e24caf14-8483-4743-b80c-ca46c28c75df"
    Publisher="CN= User1"
    Version="1.0.0.0" />
```

Locating the Application Id in the Manifest

To locate the Application Id in the manifest file for an installed UWP app, look for the *Application Id* entry.

For example, for the hello world app the Application ID is *App*.

```
<Application Id="App"
      Executable="$targetnametoken$.exe"
      EntryPoint="HelloWorld.App">
```

Example WinDbg Command Line

This is an example command line loading the HelloWorld app under the debugger using the full package name and AppId.

```
windbg.exe -plmPackage e24caf14-8483-4743-b80c-ca46c28c75df_1.0.0.0_x86__97ghe447vaan8 -plmApp App
```

Launching a background task under the debugger

A background task can be explicitly launched under the debugger from the command line using the TaskId. To do this, use the -plmPackage and -plmBgTaskId command line parameters:

```
windbg.exe -plmPackage <PLMPackageName> -plmBgTaskId <BackgroundTaskId>
```

Parameter	Description
<PLMPackageName>	The name of the application package. Use the .querypackages command to lists all UWP applications. Do not provide a path to the location of the package, provide just the package name.
<BackgroundTaskId>	The BackgroundTaskId can be located using the .querypackages command as described below. For more information about the application manifest file, see App package manifest.

This is an example of loading the SDKSamples.BackgroundTask code under the debugger.

```
windbg.exe -plmPackage Microsoft.SDKSamples.BackgroundTask.CPP_1.0.0.0_x64__8wekyb3d8bbwe -plmBgTaskId
{ee4438ee-22db-4cdd-85e4-8ad8a1063523}
```

You can experiment with the Background task sample code to become familiar with UWP debugging. It can be downloaded at Background task sample.

Use the .querypackages command to locate the BackgroundTaskId. Use CTRL-F to locate the app and then locate the *Background Task Id* field. The background task must be running to display the associated background task name and task Id.

```
0:000> .querypackages
...
Package Full Name: Microsoft.SDKSamples.BackgroundTask.CPP_1.0.0.0_x86__8wekyb3d8bbwe
Package Display Name: BackgroundTask C++ sample
Version: 1.0.0.0
Processor Architecture: x86
Publisher: CN=Microsoft Corporation, O=Microsoft Corporation, L=Redmond, S=Washington, C=US
Publisher Display Name: Microsoft Corporation
Install Folder: C:\Users\user1\Documents\Visual Studio
2015\Projects\Background_task_sample\C++\Debug\BackgroundTask.Windows\AppX
Package State: Running
AppId: BackgroundTask.App
Background Task Name: SampleBackgroundTask
Background Task Id: {ee4438ee-22db-4cdd-85e4-8ad8a1063523}
...
```

If you know the full package name you can use .querypackage to display the *Background Task Id* field.

You can also locate the BackgroundTaskId by using the enumerateBgTasks option of the PLMDebug. For more information about the PMLDebug utiltity, see **PLMDebug**.

```
C:\Program Files\Debugging Tools for Windows (x64)>PLMDebug /enumerateBgTasks
Microsoft.SDKSamples.BackgroundTask.CPP_1.0.0.0_x64__8wekyb3d8bbwe
Package full name is Microsoft.SDKSamples.BackgroundTask.CPP_1.0.0.0_x64__8wekyb3d8bbwe.
Background Tasks:
SampleBackgroundTask : {C05806B1-9647-4765-9A0F-97182CEA5AAD}

SUCCEEDED
```

Debugging a UWP process remotely using a Process Server (DbgSrv)

All of the -plm* commands work correctly with dbgsrv. To debug using dbgsrv, use the -premote switch with the connection string for dbgsrv:

```
windbg.exe -premote npipe:pipe=fdsa,server=localhost -plmPackage e24caf14-8483-4743-b80c-
ca46c28c75df_1.0.0.0_x86__97ghe447vaan8 -plmApp App
```

For more information about the -premote options, see Process Servers (User Mode) and Process Server Examples.

Summary of UWP app commands

This section provides a summary of UWP app debugger commands

Gathering Package Information

.querypackage

The .querypackage displays the state of a UWP application. For example, if the app is running, it can be in the *Active* state.

```
.querypackage <PLMPackageName>
```

Example:

```
0:000> .querypackage e24caf14-8483-4743-b80c-ca46c28c75df_1.0.0.0_x86__97ghe447vaan8
Package Full Name: e24caf14-8483-4743-b80c-ca46c28c75df_1.0.0.0_x86__97ghe447vaan8
Package Display Name: HelloWorld
Version: 1.0.0.0
Processor Architecture: x86
Publisher: CN=user1
Publisher Display Name: user1
Install Folder: c:\users\user1\documents\visual studio
2015\Projects\HelloWorld\HelloWorld\bin\x86\Release\AppX
Package State: Running
AppId: App
Executable: HelloWorld.exe
```

.querypackages

The .querypackages command lists all the installed UWP applications and their current state.

```
.querypackages
```

Example:

```
0:000> .querypackages
...
Package Full Name: Microsoft.MicrosoftSolitaireCollection_3.9.5250.0_x64__8wekyb3d8bbwe
Package Display Name: Microsoft Solitaire Collection
Version: 3.9.5250.0
Processor Architecture: x64
Publisher: CN=Microsoft Corporation, O=Microsoft Corporation, L=Redmond, S=Washington, C=US
Publisher Display Name: Microsoft Studios
Install Folder: C:\Program
Files\WindowsApps\Microsoft.MicrosoftSolitaireCollection_3.9.5250.0_x64__8wekyb3d8bbwe
Package State: Unknown
AppId: App

Package Full Name: e24caf14-8483-4743-b80c-ca46c28c75df_1.0.0.0_x86__97ghe447vaan8
Package Display Name: HelloWorld
Version: 1.0.0.0
Processor Architecture: x86
Publisher: CN=user1
Publisher Display Name: user1
Install Folder: c:\users\user1\documents\visual studio
2015\Projects\HelloWorld\HelloWorld\bin\x86\Release\AppX
Package State: Running
AppId: App
Executable: HelloWorld.exe

Package Full Name: Microsoft.SDKSamples.BackgroundTask.CPP_1.0.0.0_x86__8wekyb3d8bbwe
Package Display Name: BackgroundTask C++ sample
Version: 1.0.0.0
Processor Architecture: x86
Publisher: CN=Microsoft Corporation, O=Microsoft Corporation, L=Redmond, S=Washington, C=US
Publisher Display Name: Microsoft Corporation
Install Folder: C:\Users\user1\Documents\Visual Studio
2015\Projects\Background_task_sample\C++\Debug\BackgroundTask.Windows\AppX
Package State: Unknown
AppId: BackgroundTask.App

...
```

Launching an app for Debugging

.createpackageapp

The .createpackageapp command enables debugging and launches a UWP application.

```
.createpackageapp <PLMPackageName> <ApplicationId> [<parameters>]
```

This table lists the parameters for .createpackageapp.

Parameter	Description
<PLMPackageName>	The name of the application package. Use the .querypackages command to lists all UWP applications. Do not provide a path to the location of the package, provide just the package name.
<ApplicationId>	The ApplicationId can be located using .querypackage or .querypackages as discussed earlier in this topic. For more information about the application manifest file, see App package manifest.

[<parameters>]	Optional parameters that are passed to the application. Not all applications require or use these optional parameters.

Example:

```
.createpackageapp e24caf14-8483-4743-b80c-ca46c28c75df_1.0.0.0_x86__97ghe447vaan8 App
```

Enabling and Disabling Use of Debug Commands

.enablepackagedebug

The .enablepackagedebug command enables debugging for UWP application. You must use .enablepackagedebug before you call any of the suspend, resume, or terminate functions.

Note that the .createpackageapp command also enables debugging of the app.

```
.enablepackagedebug <PLMPackageName>
```

Example:

```
.enablepackagedebug e24caf14-8483-4743-b80c-ca46c28c75df_1.0.0.0_x86__97ghe447vaan8
```

.disablepackagedebug

The .disablepackagedebug command disables debugging for UWP application.

```
.disablepackagedebug <PLMPackageName>
```

Example:

```
.disablepackagedebug e24caf14-8483-4743-b80c-ca46c28c75df_1.0.0.0_x86__97ghe447vaan8
```

Starting and Stopping apps

Note that suspend, resume, and terminate affect all currently running apps in the package.

.suspendpackage

The .suspendpackage command, suspends a UWP application.

```
.suspendpackage <PLMPackageName>
```

Example:

```
0:024> .suspendpackage e24caf14-8483-4743-b80c-ca46c28c75df_1.0.0.0_x86__97ghe447vaan8
```

.resumepackage

The .resumepackage command resumes a UWP application.

```
.resumepackage <PLMPackageName>
```

Example:

```
.resumepackage e24caf14-8483-4743-b80c-ca46c28c75df_1.0.0.0_x86__97ghe447vaan8
```

.terminatepackageapp

The .terminatepackageapp command terminates the all of the UWP applications in the package.

```
.terminatepackageapp <PLMPackageName>
```

Example:

```
.terminatepackageapp e24caf14-8483-4743-b80c-ca46c28c75df_1.0.0.0_x86__97ghe447vaan8
```

Background Tasks

.activatepackagebgtask

The .activatepackagebgtask command enables debugging and launches a UWP background task.

```
 .activatepackagebgtask <PLMPackageName> <bgTaskId>
```

Example:

```
.activatepackagebgtask Microsoft.SDKSamples.BackgroundTask.CPP_1.0.0.0_x64__8wekyb3d8bbwe {C05806B1-9647-4765-
9A0F-97182CEA5AAD}
```

Usage Examples

Attach a debugger when your app is launched

Suppose you have an app named HelloWorld that is in a package named e24caf14-8483-4743-b80c-ca46c28c75df_1.0.0.0_x86__97ghe447vaan8. Verify that your package is installed by displaying the full names and running states all installed packages. In a Command Prompt window, enter the following command. You can use CTRL+F to search the command output for the app name of HelloWorld.

```
.querypackages
...

Package Full Name: e24caf14-8483-4743-b80c-ca46c28c75df_1.0.0.0_x86__97ghe447vaan8
Package Display Name: HelloWorld
Version: 1.0.0.0
Processor Architecture: x86
Publisher: CN=user1
Publisher Display Name: user1
Install Folder: c:\users\user1\documents\visual studio
2015\Projects\HelloWorld\HelloWorld\bin\x86\Release\AppX
Package State: Unknown
AppId: App

...
```

Use .createpackageapp to launch and attach to the app. The .createpackageapp command also enables debugging of the app.

```
.createpackageapp e24caf14-8483-4743-b80c-ca46c28c75df_1.0.0.0_x86__97ghe447vaan8 App
```

When you have finished debugging, decrement the debug reference count for the package using the .disablepackagedebug command.

```
.disablepackagedebug e24caf14-8483-4743-b80c-ca46c28c75df_1.0.0.0_x86__97ghe447vaan8
```

Attach a debugger to an app that is already running

Suppose you want to attach WinDbg to MyApp, which is already running. In WinDbg, on the **File** menu, choose **Attach to a Process**. Note the process ID for MyApp. Let's say the process ID is 4816. Increment the debug reference count for the package that contains MyApp.

```
.enablepackagedebug e24caf14-8483-4743-b80c-ca46c28c75df_1.0.0.0_x86__97ghe447vaan8
```

In WinDbg, in the **Attach to Process** dialog box, select process 4816, and click OK. WinDbg will attach to MyApp.

When you have finished debugging, decrement the debug reference count for the package using the .disablepackagedebug command.

```
.disablepackagedebug e24caf14-8483-4743-b80c-ca46c28c75df_1.0.0.0_x86__97ghe447vaan8
```

Manually suspend and resume your app

Follow these steps to manually suspend and resume your app. First, increment the debug reference count for the package that contains your app.

```
.enablepackagedebug  e24caf14-8483-4743-b80c-ca46c28c75df_1.0.0.0_x86__97ghe447vaan8
```

Suspend the package. Your app's suspend handler is called, which can be helpful for debugging.

```
.suspendpackage e24caf14-8483-4743-b80c-ca46c28c75df_1.0.0.0_x86__97ghe447vaan8
```

When you have finished debugging, resume the package.

```
.resumepackage e24caf14-8483-4743-b80c-ca46c28c75df_1.0.0.0_x86__97ghe447vaan8
```

Finally, decrement the debug reference count for the package.

```
.disablepackagedebug e24caf14-8483-4743-b80c-ca46c28c75df_1.0.0.0_x86__97ghe447vaan8
```

Related topics

Debugging Using WinDbg

Opening a Dump File Using WinDbg

12/5/2018 • 2 minutes to read • Edit Online

There are several ways you can use WinDbg to open a dump file.

WinDbg Menu

If WinDbg is already running and is in dormant mode, you can open a dump by choosing **Open Crash Dump** from the **File** menu or by pressing CTRL+D. When the Open Crash Dump dialog box appears, enter the full path and name of the crash dump file in the **File name** box, or use the dialog box to select the proper path and file name. When the proper file has been chosen, click **Open**.

Command Prompt

In a Command Prompt window, you can open a dump file when you launch WinDbg. Use the following command:

windbg -y *SymbolPath* **-i** *ImagePath* **-z** *DumpFileName*

The **-v** option (verbose mode) is also useful. For more information about the command-line syntax, see **WinDbg Command-Line Options**.

Debugger Command Window

If WinDbg is already in a kernel-mode debugging session, you can open a dump file by using the **.opendump (Open Dump File)** command, followed by **g (Go)**.

Live Kernel-Mode Debugging Using WinDbg

12/5/2018 • 5 minutes to read • Edit Online

There are two ways you can use WinDbg to initiate a live kernel-mode debugging session.

WinDbg Menu

When WinDbg is in dormant mode, you can begin a kernel debugging session by choosing **Kernel Debug** from the **File** menu or by pressing CTRL+K. When the **Kernel Debugging** dialog box appears, click the appropriate tab: **NET**, **1394**, **USB**, **COM**, or **Local**. Each tab specifies a different connection method. For more information about the dialog box and its entries, see File | Kernel Debug.

Command Prompt

In a Command Prompt window, you can initiate a kernel-mode debugging session when you launch WinDbg. Enter one of the following commands:

windbg [-y *SymbolPath*] -k net:port=*PortNumber*,key=*Key*[,target=*TargetIPAddress*|*TargetMachineName*]

windbg [-y *SymbolPath*] -k 1394:channel=*1394Channel*[,symlink=*1394Protocol*]

windbg [-y *SymbolPath*] -k usb:targetname=*USBString*

windbg [-y *SymbolPath*] -k com:port=*ComPort*,baud=*BaudRate*

windbg [-y *SymbolPath*] -k com:pipe,port=*VMHost*\pipe*PipeName*[,resets=0][,reconnect]

windbg [-y *SymbolPath*] -k com:*modem*

windbg [-y *SymbolPath*] -kl

windbg [-y *SymbolPath*] -k

For more information, see **WinDbg Command-Line Options**.

Environment Variables

For debugging over a serial (COM port) or 1394 connection, you can use environment variables to specify the connection settings.

Use the following variables to specify a serial connection.

set _NT_DEBUG_PORT = *ComPort*

set _NT_DEBUG_BAUD_RATE = *BaudRate*

Use the following variables to specify a 1394 connection.

set _NT_DEBUG_BUS = *1394*

set _NT_DEBUG_1394_CHANNEL = *1394Channel*

set _NT_DEBUG_1394_SYMLINK = *1394Protocol*

For more information, see Kernel-Mode Environment Variables.

Parameters

SymbolPath
A list of directories where symbol files are located. Directories in the list are separated by semicolons. For more information, see Symbol Path.

PortNumber
A port number to use for network debugging. You can choose any number from 49152 through 65535. For more information, see Setting Up a Network Connection Manually.

Key
The encryption key to use for network debugging. We recommend that you use an automatically generated key, which is provided by bcdedit when you configure the target computer. For more information, see Setting Up a Network Connection Manually.

TargetIPAddress
The IPv4 address of the target machine.

When the target= IP address is specified, this causes the debugger to initiate a connection to the specified target machine, by sending a special packet to the target, that will cause it to attempt to connect with that debugger. The debugger will send packets to the target repeatedly approximately every half second, attempting to connect. If the connection is successful, the target will drop any existing connection, and communicate only with this instance of the debugger. This allows you to take control of the debugging session away from an existing debugging connection.

When the target is configured with a host IP address, and the debugger is being run on the machine with the configured host IP address, there is no need to specify the target= IP address parameter. When the target is configured with a host IP address, it will send OFFER packets to the host every three seconds. The OFFER packets allow the debugger to connect to the host when no target= IP address is specified.

For more information on configuring the host IP address on the target, see Setting Up KDNET Network Kernel Debugging Automatically and Setting Up KDNET Network Kernel Debugging Manually.

TargetMachineName
The machine name of the target PC. To use the machine name, the DNS system on the network must have the machine name associated with the IP address of the target PC.

1394Channel
The 1394 channel number. Valid channel numbers are any integer between 0 and 62, inclusive. *1394Channel* must match the number used by the target computer, but does not depend on the physical 1394 port chosen on the adapter. For more information, see Setting Up a 1394 Connection Manually.

1394Protocol
The connection protocol to be used for the 1394 kernel connection. This can almost always be omitted, because the debugger will automatically choose the correct protocol. If you wish to set this manually, and the target computer is running Windows XP, *1394Protocol* should be set equal to "channel". If the target computer is running Windows Server 2003 or later, *1394Protocol* should be set equal to "instance". If it is omitted, the debugger will default to the protocol appropriate for the current target computer. This can only be specified through the command line or the environment variables, not through the WinDbg graphical interface.

USBString
A USB connection string. This must match the string specified with the /targetname boot option. For more information, see Setting Up a USB 3.0 Connection Manually and Setting Up a USB 2.0 Connection Manually.

ComPort
The name of the COM port. This can be in the format "com2" or in the format "\\.\com2", but should not simply be a number. For more information, see Setting Up a Serial Connection Manually.

BaudRate
The baud rate. This can be 9600, 19200, 38400, 57600, or 115200.

VMHost
When debugging a virtual machine, *VMHost* specifies the name of the physical computer on which the virtual machine is running. If the virtual machine is running on the same computer as the kernel debugger itself, use a single period (.) for *VMHost*. For more information, see Setting Up a Connection to a Virtual Machine.

PipeName
The name of the pipe created by the virtual machine for the debugging connection.

resets=0
Specifies that an unlimited number of reset packets can be sent to the target when the host and target are synchronizing. This parameter is only needed when debugging certain kinds of virtual machines.

reconnect
Causes the debugger to automatically disconnect and reconnect the pipe if a read/write failure occurs. Additionally, if the named pipe is not found when the debugger is started, the reconnect parameter will cause it to wait for a pipe of this name to appear. This parameter is only needed when debugging certain kinds of virtual machines.

-kl
Causes the debugger to perform local kernel-mode debugging. For more information, see Local Kernel-Mode Debugging.

Examples

The following batch file could be used to set up and start a debugging session over a COM port connection.

```
set _NT_SYMBOL_PATH=d:\mysymbols
set _NT_DEBUG_PORT=com1
set _NT_DEBUG_BAUD_RATE=115200
set _NT_DEBUG_LOG_FILE_OPEN=d:\debuggers\logfile1.log
windbg -k
```

The following batch file could be used to set up and start a debugging session over a 1394 connection.

```
set _NT_SYMBOL_PATH=d:\mysymbols
set _NT_DEBUG_BUS=1394
set _NT_DEBUG_1394_CHANNEL=44
set _NT_DEBUG_LOG_FILE_OPEN=d:\debuggers\logfile1.log
windbg -k
```

The following command lines could be used to start WinDbg without any environment variables.

windbg -y d:\mysymbols -k com:port=com2,baud=57600

windbg -y d:\mysymbols -k com:port=\\.\com2,baud=115200

windbg -y d:\mysymbols -k 1394:channel=20,symlink=instance

windbg -y d:\mysymbols -k net:port=50000,key=AutoGeneratedKey

windbg -y d:\mysymbols -k net:port=50000,key=AutoGeneratedKey,target=TargetIPAddress

Related topics

WinDbg Command-Line Options

Ending a Debugging Session in WinDbg

12/5/2018 • 2 minutes to read • Edit Online

Exiting WinDbg

You can exit WinDbg by choosing **Exit** from the **File** menu or by pressing ALT+F4.

If you are performing user-mode debugging, these commands close the application that you are debugging, unless you used the **-pd** command-line option when you started the debugger.

If you are performing kernel-mode debugging, the target computer remains in its current state. This situation enables you to leave the target running or frozen. (If you leave the target frozen, any future connection from a kernel debugger can resume debugging where you left it.)

Ending a User-Mode Session Without Exiting

To end a user-mode debugging session, return the debugger to dormant mode, and close the target application, you can use the following methods:

- Enter the **.kill (Kill Process)** command.
- Enter the **q (Quit)** command (unless you started the debugger with the **-pd** option).
- Choose **Stop Debugging** from the **Debug** menu.
- Press SHIFT+F5.
- Click the **Stop Debugging** button ( ) on the toolbar

To end a user-mode debugging session, return the debugger to dormant mode, and set the target application running again, you can use the following methods:

- Enter the **.detach (Detach from Process)** command. If you are debugging multiple targets, this command detaches from the current target and continues the debugging session with the remaining targets.
- Choose **Detach Debugee** from the **Debug** menu. If you are debugging multiple targets, this command detaches from all current targets.
- Enter the **qd (Quit and Detach)** command.
- Enter the **q (Quit)** command, if you started the debugger with the **-pd** option.

To end a user-mode debugging session, return the debugger to dormant mode, but leave the target application in the debugging state, you can use the following method:

- Enter the **.abandon (Abandon Process)** command.

For information about reattaching to the target, see Reattaching to the Target Application.

Ending a Kernel-Mode Session Without Exiting

To end a kernel-mode debugging session, return the debugger to dormant mode, and leave the target computer frozen, you can use the following methods:

- Enter the **q (Quit)** command (unless you started the debugger with the **-pd** option)

- Choose **Stop Debugging** from the **Debug** menu.
- Press SHIFT+F5.
- Click the **Stop debugging (Shift+F5)** button ( ) on the toolbar.

When a WinDbg session ends, you are prompted to save the workspace for the current session, and then WinDbg returns to dormant mode. At this point, you can use all starting options. That is, you can start to debug a running process, spawn a new process, attach to a target computer, open a crash dump, or connect to a remote debugging session.

Setting Symbol and Executable Image Paths in WinDbg

12/5/2018 • 3 minutes to read • Edit Online

Symbol Path

The symbol path specifies the directories where the symbol files are located. For more information about symbols and symbol files, see Symbols.

Note If you are connected to the Internet or a corporate network, the most efficient way to access symbols is to use a symbol server. You can use a symbol server by using the srv* or symsrv* string within your symbol path. For more information about symbol servers, see Symbol Stores and Symbol Servers.

To control the symbol path in WinDbg, do one of the following:

- Choose **Symbol File Path** from the **File** menu or press CTRL+S.
- Use the **.sympath (Set Symbol Path)** command. If you are using a symbol server, the **.symfix (Set Symbol Store Path)** command is similar to **.sympath** but saves you typing.
- When you start the debugger, use the **-y** command-line option. See **WinDbg Command-Line Options**.
- Before you start the debugger, use the _NT_SYMBOL_PATH and _NT_ALT_SYMBOL_PATH environment variables to set the path. The symbol path is created by appending _NT_SYMBOL_PATH after _NT_ALT_SYMBOL_PATH. (Typically, the path is set through the _NT_SYMBOL_PATH. However, you might want to use _NT_ALT_SYMBOL_PATH to override these settings in special cases, such as when you have private versions of shared symbol files.) If you try to add an invalid directory through these environment variables, the debugger ignores this directory.

  Note If you use the -**sins** command-line option, the debugger ignores the symbol path environment variable.

Executable Image Path

An executable file is a binary file that the processor can run. These files typically have the .exe, .dll, or .sys file name extension. Executable files are also known as modules, especially when executable files are described as units of a larger application. Before the Windows operating system runs an executable file, it loads it into memory. The copy of the executable file in memory is called the executable image or the image.

Note These terms are sometimes used imprecisely. For example, some documents might use "image" for the actual file on the disk. Also, the Windows kernel and HAL have special module names. For example, the **nt** module corresponds to the Ntoskrnl.exe file.

The executable image path specifies the directories that the binary executable files are located in.

In most situations, the debugger knows the location of the executable files, so you do not have to set the path for this file.

However, there are situations when this path is required. For example, kernel-mode small memory dump files do not contain all of the executable files that exist in memory at the time of a stop error (that is, a crash). Similarly, user-mode minidump files do not contain the application binaries. If you set the path of the executable files, the debugger can find these binary files.

The debugger's executable image path is a string that consists of multiple directory paths, separated by semicolons. Relative paths are supported. However, unless you always start the debugger from the same directory, you should add a drive letter or a network share before each path. Network shares are also supported. The debugger searches the executable image path recursively. That is, the debugger searches the subdirectories of each directory that is listed in this path.

To control the executable image path in WinDbg, do one of the following:

- Choose **Image File Path** from the **File** menu, or press CTRL+I.
- Use the **.exepath (Set Executable Path)** command.
- When you start the debugger, use the **-i** command-line option. See **WinDbg Command-Line Options**.
- Before you start the debugger, use the _NT_EXECUTABLE_IMAGE_PATH environment variable to set the path.

  Note If you use the **-sins** command-line option, the debugger ignores the executable image path environment variable.

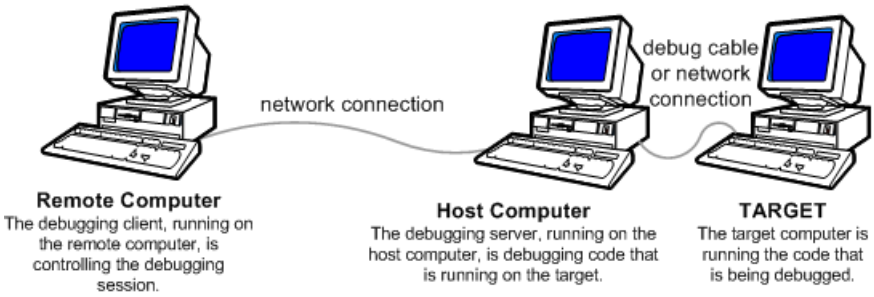
Remote Debugging Using WinDbg

12/5/2018 • 2 minutes to read • Edit Online

Remote debugging involves two debuggers running at two different locations. The debugger that performs the debugging is called the *debugging server*. The second debugger, called the *debugging client*, controls the debugging session from a remote location. To establish a remote session, you must set up the debugging server first and then activate the debugging client.

The code that is being debugged could be running on the same computer that is running the debugging server, or it could be running on a separate computer. If the debugging server is performing user-mode debugging, then the process that is being debugged can run on the same computer as the debugging server. If the debugging server is performing kernel-mode debugging, then the code being debugged would typically run on a separate target computer.

The following diagram illustrates a remote session where the debugging server, running on a host computer, is performing kernel-mode debugging of code that is running on a separate target computer.



There are several transport protocols you can use for a remote debugging connection: TCP, NPIPE, SPIPE, SSL, and COM Port. Suppose you have chosen to use TCP as the protocol and you have chosen to use WinDbg as both the debugging client and the debugging server. You can use the following procedure to establish a remote kernel-mode debugging session:

1. On the host computer, open WinDbg and establish a kernel-mode debugging session with a target computer. (See Live Kernel-Mode Debugging Using WinDbg.)
2. Break in by choosing **Break** from the **Debug** menu or by pressing CTRL-Break.
3. In the Debugger Command Window, enter the following command.

   .server tcp:port=5005

   Note The port number 5005 is arbitrary. The port number is your choice.
4. WinDbg will respond with output similar to the following.

   ```
   Server started.  Client can connect with any of these command lines
   0: <debugger> -remote tcp:Port=5005,Server=YourHostComputer
   ```
5. On the remote computer, open WinDbg, and choose **Connect to Remote Session** from the **File** menu.
6. Under **Connection String**, enter the following string.

   tcp:Port=5005,Server=*YourHostComputer*

   where *YourHostComputer* is the name of your host computer, which is running the debugging server.

Click **OK**.

Using the Command Line

As an alternative to the procedure given in the preceding section, you can set up a remote debugging session at the command line. Suppose you are set up to establish a kernel-mode debugging session, between a host computer and a target computer, over a 1394 cable on channel 32. You can use the following procedure to establish a remote debugging session:

1. On the host computer, enter the following command in a Command Prompt window.

   windbg -server tcp:port=5005 -k 1394:channel=32

2. On the remote computer, enter the following command in a Command Prompt window.

   windbg -remote tcp:Port=5005,Server=*YourHostComputer*

   where *YourHostComputer* is the name of your host computer, which is running the debugging server.

Additional Information

There are many ways to establish remote debugging other than the ones shown in this topic. For complete information about setting up a debugging server in the WinDbg Debugger Command Window, see **.server (Create Debugging Server)**. For complete information about launching WinDbg (and establishing remote debugging) at the command line, see **WinDbg Command-Line Options**.

Entering Debugger Commands in WinDbg

12/5/2018 • 2 minutes to read • Edit Online

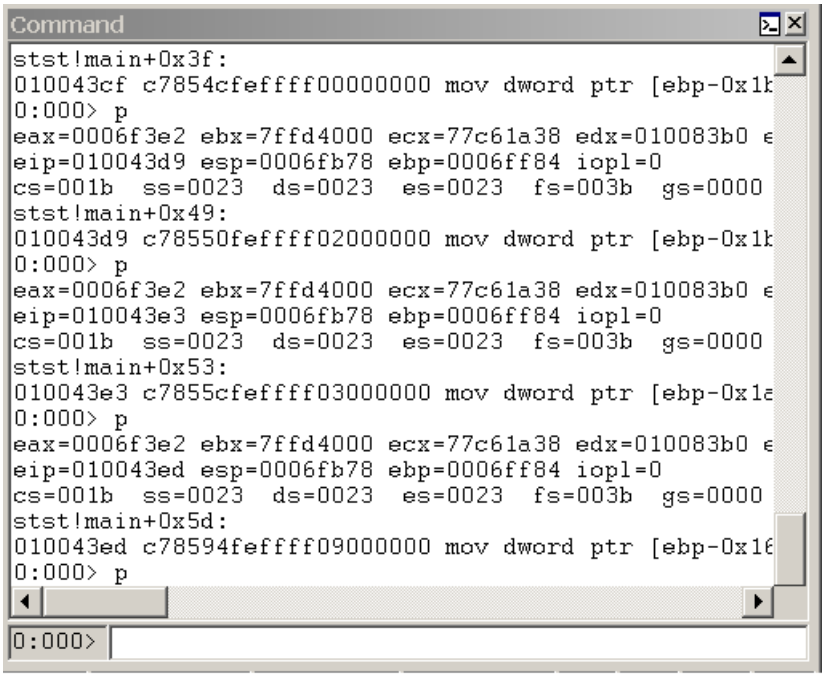
The Debugger Command window is the primary debugging information window in WinDbg. You can enter debugger commands and view the command output in this window.

Note This window displays "Command" in the title bar. However, this documentation always refers to this window as "the Debugger Command window" to avoid confusing it with the Command Prompt windows that are used to issue Microsoft MS-DOS commands.

Opening the Debugger Command Window

To open the Debugger Command window, choose **Command** from the **View** menu. (You can also press ALT+1 or click the **Command** button ( ) on the toolbar. ALT+SHIFT+1 closes the Debugger Command window.)

The following screen shot shows an example of a Debugger Command window.



Using the Debugger Command Window

The Debugger Command window is split into two panes. You type commands in the smaller pane (the command entry pane) at the bottom of the window and view the output in the larger pane at the top of the window.

In the command entry pane, use the UP ARROW and DOWN ARROW keys to scroll through the command history. When a command appears, you can edit it or press ENTER to run the command.

The Debugger Command window contains a shortcut menu with additional commands. To access this menu, right-click the title bar of the window or click the icon near the upper-right corner of the window ( ). The following list describes some of the menu commands:

- **Add to command output** adds a comment to the command output, similar to the Edit | Add to Command Output command.
- **Clear command output** deletes all of the text in the window.

- **Choose text color and recolor selection...** opens a dialog box that enables you to choose the text color in which to display the text that is selected in the Debugger Command window.
- **Word wrap** turns the word wrap status on and off. This command affects the whole window, not only commands that you use after this state is selected. Because many commands and extensions produce formatted displays, it is not recommended that you use word wrap.
- **Mark current location** sets a marker at the current cursor location in the command window. The name of the mark is the contents of the line to the right of the cursor.
- **Go to mark** causes the window to scroll so that the line that contains the chosen mark is positioned at the top of the window.
- **Always floating** causes the window to remain undocked, even if it is dragged to a docking location.
- **Move with frame** causes the window to move when the WinDbg frame is moved, even if the window is undocked. For more information about docked, tabbed, and floating windows, see Positioning the Windows.

Using the Command Browser Window in WinDbg

12/5/2018 • 2 minutes to read • Edit Online

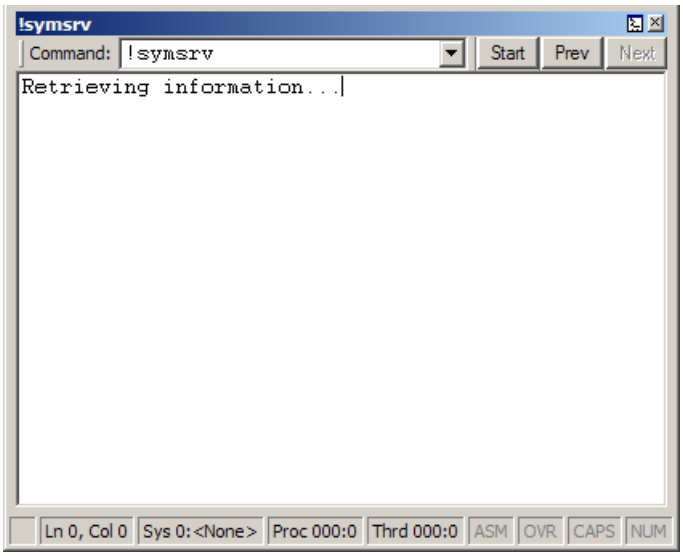
A Command Browser window displays and stores the text results of a debugger command. This window creates a command reference that enables you to view the results of a specific command without re-entering the command. A Command Browser window also provides navigation through the stored commands, so you can more quickly access commands than with the Debugger Command window.

Opening the Command Browser Window

You can open multiple Command Browser windows at one time. To open a Command Browser window, choose **Command Browser** from the **View** menu. (You can also press CTRL+N or click the **Command Browser** button ( ) on the toolbar. ALT+SHIFT+N closes the Command Browser window.)

You can also open a Command Browser window by entering **.browse (Display Command in Browser)** in the regular Debugger Command window.

The following screen shot shows an example of a Command Browser window.



Using the Command Browser Window

In the Command Browser window, you can do the following:

- To enter a command, type it in the **Command** box.
- To view the results of a previously entered command, use the **Start**, **Prev**, and **Next** buttons to scroll through the command list, or select one of the preceding 20 commands from the **Command** menu. To find a command that is not one of the preceding 20 commands, use the **Next** button.

The Command Browser window has a shortcut menu with additional commands. To access the menu, right-click the title bar or click the icon near the upper-right corner of the window ( ). The following list describes some of the menu commands:

- **Start**, **Prev**, and **Next** move the cursor to the start of the command history or to the previous or next command, respectively.
- **Add to Recent Commands** puts the current command into the **Recent Commands** menu of the **View** menu in the WinDbg window. Recent commands are saved in the workspace.
- **Toolbar** turns the toolbar on and off.

- **Move to new dock** closes the Command Browser window and opens it in a new dock.
- **Always floating** causes the window to remain undocked even if it is dragged to a docking location.
- **Move with frame** causes the window to move when the WinDbg frame is moved, even if the window is undocked. For more information about docked, tabbed, and floating windows, see Positioning the Windows.

Commands that you enter in a Command Browser window are executed by the debugger engine, not by the WinDbg user interface. This means that you cannot enter user interface commands like **.cls** in a Command Browser window. If the user interface is a remote client, the server (not the client) executes the command.

A command that you enter in a Command Browser window executes synchronously, so it does not display output until it has completed.

Command Browser windows are saved in the WinDbg workspace, but the command histories are not saved. Only the current command for each Command Browser window is saved in the workspace.

Setting Breakpoints in WinDbg

12/5/2018 • 2 minutes to read • Edit Online

There are several ways you can set, view, and manipulate breakpoints using WinDbg.

Debugger Command Window

You can set, view, and manipulate breakpoints by entering commands in the Debugger Command Window. For a list of commands, see Methods of Controlling Breakpoints.

WinDbg Menu

You can open the **Breakpoints** dialog box by choosing **Breakpoints** from the **Edit** menu or by pressing ALT+F9. This dialog box lists all breakpoints, and you can use it to disable, enable, or clear existing breakpoints or to set new breakpoints.

Code Windows

The Disassembly window and the Source windows highlight lines that have breakpoints set. Enabled breakpoints are red, and disabled breakpoints are yellow.

If you set the cursor on a specific line in the Disassembly window or in a Source window, you can press F9 to set a breakpoint at that line.

Viewing the Call Stack in WinDbg

12/5/2018 • 2 minutes to read • Edit Online

The call stack is the chain of function calls that have led to the current location of the program counter. The top function on the call stack is the current function, the next function is the function that called the current function, and so on. The call stack that is displayed is based on the current program counter, unless you change the register context. For more information about how to change the register context, see Changing Contexts.

In WinDbg, you can view the call stack by entering commands or by using the Calls window.

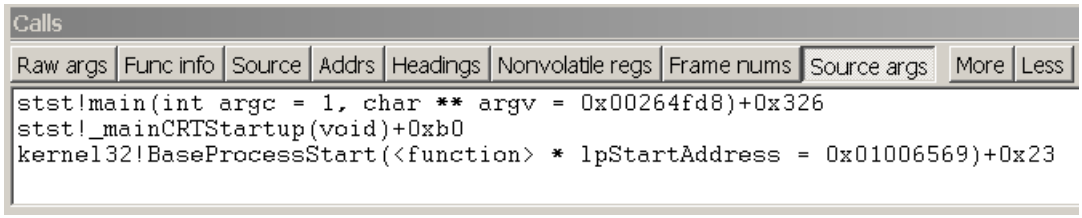
Debugger Command Window

You can view the call stack by entering one of the **k (Display Stack Backtrace)** commands in the Debugger Command window.

Calls Window

As an alternative to the **k** command, you can view the call stack in the Calls window. To open the Calls window, choose **Call Stack** from the **View** menu.

The following screen shot shows an example of a Calls window.



Buttons in the Calls window enable you to customize the view of the call stack. To move to the corresponding call location in the Source window or Disassembly window, double-click a line of the call stack, or select a line and press ENTER. This action also changes the local context to the selected stack frame. For more information about running to or from this point, see Controlling the Target.

In user mode, the stack trace is based on the stack of the current thread. For more information about the stack of the current thread, see Controlling Processes and Threads.

In kernel mode, the stack trace is based on the current register context. You can set the register context to match a specific thread, context record, or trap frame. For more information about setting the register context, see Register Context.

The Calls window has a toolbar that contains several buttons and has a shortcut menu with additional commands. To access this menu, right-click the title bar or click the icon near the upper-right corner of the window ( ). The toolbar and menu contain the following buttons and commands:

- **Raw args** displays the first three parameters that are passed to the function. On an x86-based processor, this display includes the first three parameters that are passed to the function ("Args to Child").
- **Func info** displays Frame Pointer Omission (FPO) data and other internal information about the function. This command is available only on an x86-based processor.
- **Source** displays source module names and line numbers after the function names (if the debugger has this information).

- **Addrs** displays various frame-related addresses. On an x86-based processor, this display includes the base pointer for the stack frame ("ChildEBP") and the return address ("RetAddr").
- **Nonvolatile regs** displays the nonvolatile portion of the register context. This command is available only on an Itanium-based processor.
- **Frame nums** displays frame numbers. Frames are always numbered consecutively, beginning with zero.
- **Arg types** displays detailed information about the arguments that are expected and received by the functions in the stack.
- **Always floating** causes the window to remain undocked even if it is dragged to a docking location.
- **Move with frame** causes the window to move when the WinDbg frame is moved, even if the window is undocked. For more information about docked, tabbed, and floating windows, see Positioning the Windows.

Additional Information

For more information about the register context and the local context, see Changing Contexts.

Assembly Code Debugging in WinDbg

12/5/2018 • 2 minutes to read • Edit Online

In WinDbg, you can view assembly code by entering commands or by using the Disassembly window.

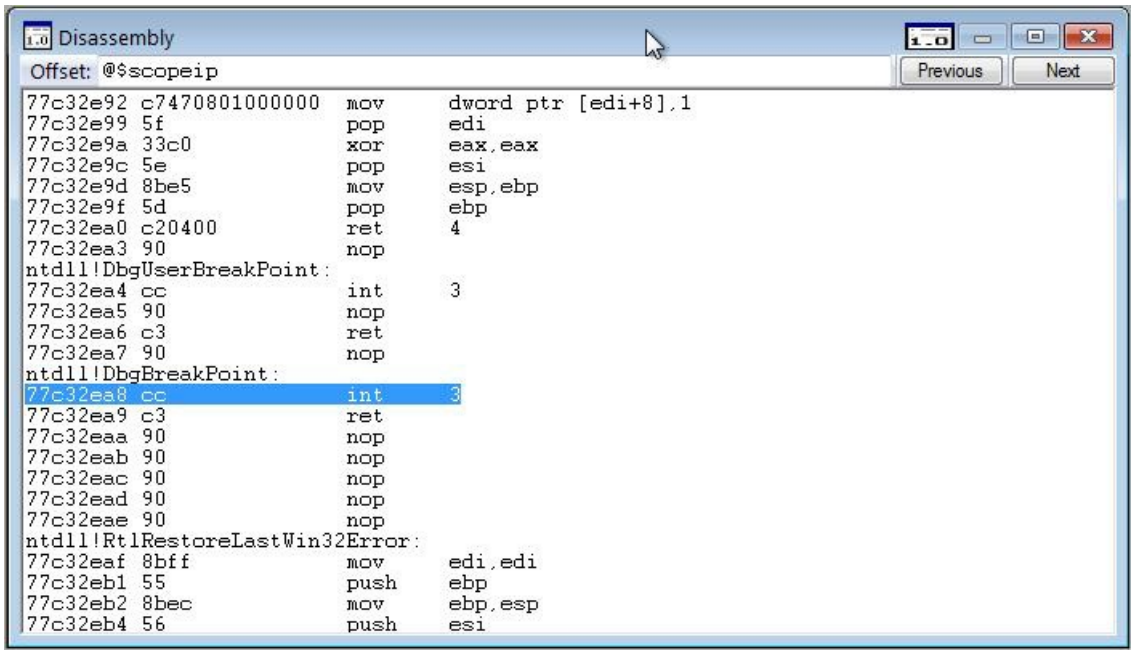
Debugger Command Window

You can view assembly code by entering the one of the **u, ub, uu (Unassemble)** commands in the Debugger Command window.

Dissasembly Window

To open or switch to the Disassembly window, choose **Dissasembly** from the **View** menu. (You can also press ALT+7 or click the **Disassembly** button ( ) on the toolbar. ALT+SHIFT+7 will close the Disassembly Window.)

The following screen shot shows an example of a Disassembly window.



The debugger takes a section of memory, interprets it as binary machine instructions, and then disassembles it to produce an assembly-language version of the machine instructions. The resulting code is displayed in the Disassembly window.

In the Disassembly window, you can do the following:

- To disassemble a different section of memory, in the **Offset** box, type the address of the memory you want to disassemble. (You can press ENTER after typing the address, but you do not have to.) The Disassembly window displays code before you have completed the address; you can disregard this code.
- To see other sections of memory, click the **Previous** or **Next** buttons or press the PAGE UP or PAGE DOWN keys. These commands display disassembled code from the preceding or following sections of memory, respectively. By pressing the RIGHT ARROW, LEFT ARROW, UP ARROW, and DOWN ARROW keys, you can navigate within the window. If you use these keys to move off of the page, a new page will appear.

The Disassembly window has a toolbar that contains two buttons and a shortcut menu with additional commands. To access the menu, right-click the title bar or click the icon that appears near the upper-right corner of the window ( ). The following list describes some of the menu commands:

- **Go to current address** opens the Source window with the source file that corresponds to the selected line in the Disassembly window and highlights this line.
- **Disassemble before current instruction** causes the current line to be placed in the middle of the Disassembly window. This command is the default option. If this command is cleared, the current line will appear at the top of the Disassembly window, which saves time because reverse-direction disassembly can be time-consuming.
- **Highlight instructions from the current source line** causes all of the instructions that correspond to the current source line to be highlighted. Often, a single source line will correspond to multiple assembly instructions. If code has been optimized, these assembly instructions might not be consecutive. This command enables you to find all of the instructions that were assembled from the current source line.
- **Show source line for each instruction** displays the source line number that corresponds to each assembly instruction.
- **Show source file for each instruction** displays the source file name that corresponds to each assembly instruction.

Additional Information

For more information about assembly debugging and related commands and a full explanation of the assembly display, see Debugging in Assembly Mode.

Source Code Debugging in WinDbg

12/5/2018 • 5 minutes to read • Edit Online

Source Path

The source path specifies the directories where the C and C++ source files are located. For more information about viewing source code in the debugger, see Source Code.

Note If you are connected to a corporate network, the most efficient way to access source files is to use a source server. You can use a source server by using the srv* string within your source path. For more information about source servers, see Using a Source Server.

To control the source path in WinDbg, do one of the following:

- Choose **Source File Path** from the **File** menu or press CTRL+P.
- Use the **.srcpath (Set Source Path)** command. If you are using a source server, **.srcfix (Use Source Server)** is slightly easier.
- Use the **.lsrcpath (Set Local Source Path)** command. If you are using a source server, **.lsrcfix (Use Local Source Server)** is slightly easier.
- When you start the debugger, use the **-srcpath** or **-lsrcpath** command-line option. See **WinDbg Command-Line Options**.
- Before you start the debugger, set the _NT_SOURCE_PATH environment variable.

Opening and Closing Source Files

To open or close a source file directly, do one of the following:

- Choose **Open Source File** from the **File** menu, or press CTRL+O. You can also use the **Open source file** button ( ) on the toolbar.

  Note When you use the menu or the toolbar button to open a source file, the path of that file is automatically appended to the source path.

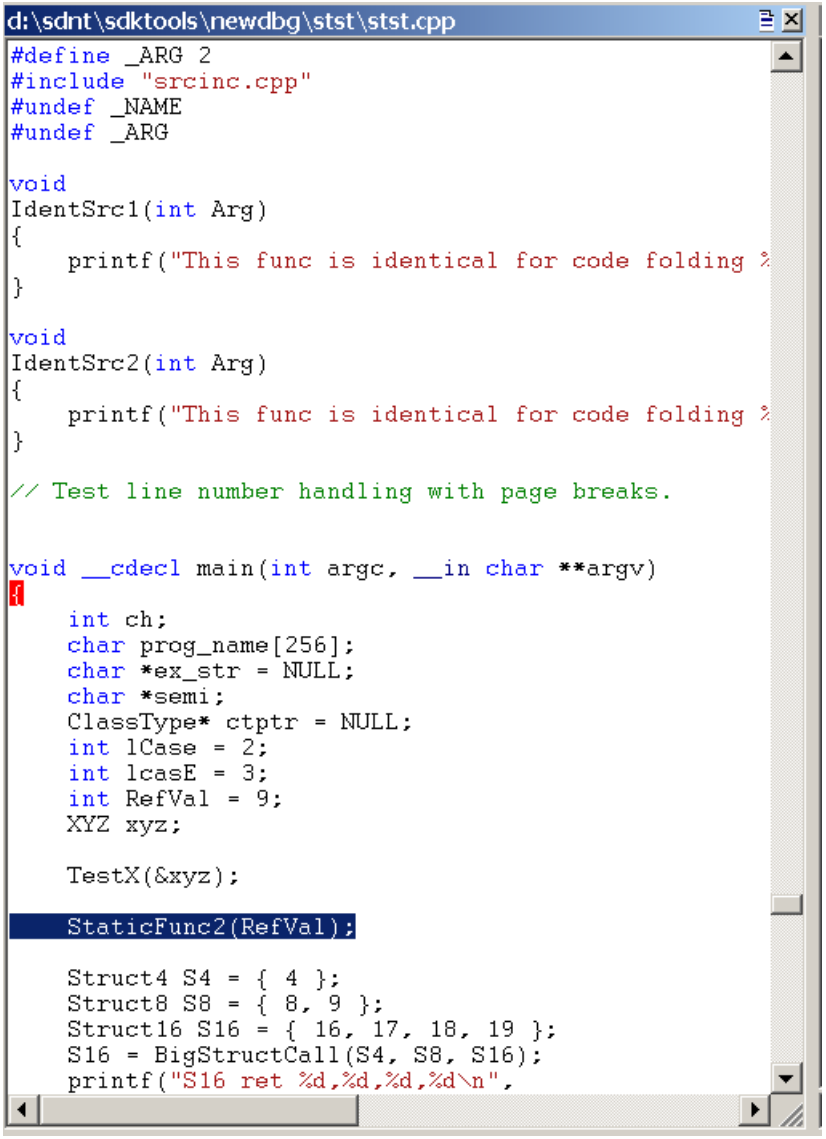
- Choose **Close Current Window** from the **File** menu.
- Click the **Close** button in the corner of the Source window.
- Choose **Recent Files** from the **File** menu to open one of the four source files that you most recently opened in WinDbg.
- Enter the **.open (Open Source File)** command.
- Enter the **lsf (Load or Unload Source File)** command.

In WinDbg, the Source window displays source files that have been loaded into the debugger.

Opening the Source Window

The debugger opens a source window when it loads a new source file. To restore or switch to an open Source window, go to the **Window** menu and choose from the list of windows at the bottom of the menu.

The following screen shot shows an example of a Source window.



Each source file resides in its own Source window. The title of each Source window is the full path of the source file.

Using the Source Window

Each Source window displays the text of one source file. You cannot edit a source file in the debugger. For more information about changing the font and tab settings, see Changing Text Properties.

Each Source window has a shortcut menu with additional commands. To access the menu, right-click the title bar or click the icon that appears near the upper-right corner of the window ( ). The following list describes some of the menu commands:

- **Set instruction pointer to current line** changes the value of the instruction pointer to the instruction that corresponds to the current line. This command is equivalent to using the Edit | Set Current Instruction command or pressing CTRL+SHIFT+I.
- **Edit this file** opens the source file in a text editor. The editor is determined by the WinDiff editor registry information or by the value of the WINDBG_INVOKE_EDITOR environment variable. For example, consider the case when the value of WINDBG_INVOKE_EDITOR is the following.

  ```
  c:\my\path\myeditor.exe -file %f -line %l
  ```

  In this case, Myeditor.exe will open to the one-based line number of the current source file. The %l option indicates that line numbers should be read as one-based, while %f indicates that the current source file should be used. Other substitution possibilities include %L, which indicates that line

numbers are zero-based, and %p, which can also indicate that the current source file should be used.

- **Evaluate selection** evaluates the currently selected text by using the C++ expression evaluator. The result appears in the Debugger Command window. If the selected text includes more than one line, a syntax error results. This command is equivalent to using the Edit | Evaluate Selection command, pressing CTRL+SHIFT+V, or using the **?? (Evaluate C++ Expression)** command with the selected text as its argument.
- **Display selected type** displays the data type of the selected object. This display appears in the Debugger Command window. If the selected text includes more than a single object, a syntax error or other irregular results might be displayed. This command is equivalent to using the Edit | Display Selected Type command or pressing CTRL+SHIFT+Y.
- **Open memory window for selection** opens a new docked Memory window that displays memory starting at the address of the selected expression.
- **Add selection to Watch window** appends the selected source token to the Watch window.
- **Disassemble at current line** causes the instruction that corresponds to the current line to appear in the Disassembly window. The selected line is highlighted in the Source window and in the Disassembly window, but this command affects only the display—the instruction pointer is not changed. If the Disassembly window is closed when this command is clicked, it is opened.
- **Select source language** displays a list of programming languages. Select the programming language that you used to generate the source file, and then click **OK** to enable basic syntax highlighting for the current Source window. Select **<None>** to disable syntax highlighting for the current Source window.

Source Window Colors and Hover Evaluation

If the debugger recognizes the source file name extension, the Source window displays certain syntax elements in color. To turn off or change the colors, do the following:

- To turn the syntax colors off in a single window, open the Source window's shortcut menu, click **Select source language**, and then click **<None>**.
- To turn the syntax colors off for all Source windows, choose **Options** from the **View** menu. Then clear the **Parse Source Languages** check box.
- To change the syntax colors, choose **Options** from the **View** menu. Then, in the **Colors** area, select a syntax element and click the **Change** button to change the color.
- The parsing method that is used for the highlighting is determined by the programming language that is associated with the file extension for the source file. To change the programming language that is associated with a specific file extension, use the File Extensions for Source Languages dialog box. To open this dialog box, choose **Source language file extensions** from the **View** menu.

The line that represents the current program counter is highlighted. Lines at which breakpoints are set are highlighted as well.

If you select a Source window and then use the mouse to hover over a symbol in that window, the symbol will be evaluated. The evaluation is the same as that produced by the **dt (Display Type)** command. To deactivate this evaluation, choose **Options** from the **View** menu. Then clear the **Evaluate on hover** check box.

Additional Information

For more information about source debugging and related commands, see Debugging in Source Mode.

Viewing and Editing Memory in WinDbg

12/5/2018 • 6 minutes to read • Edit Online

In WinDbg, you can view and edit memory by entering commands or by using a Memory window.

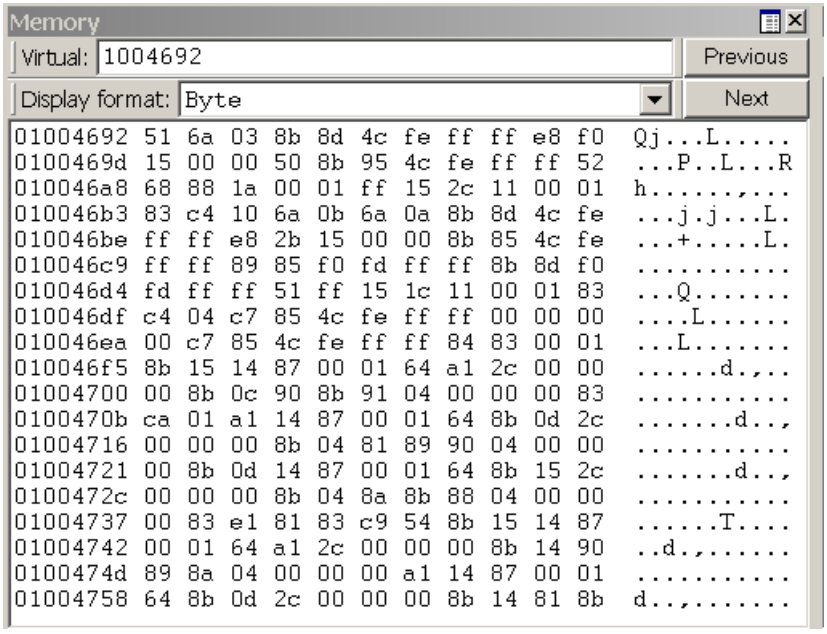
Debugger Command Window

You can view memory by entering one of the **Display Memory** commands in the Debugger Command window. You can edit memory by entering one of the **Enter Values** commands in the Debugger Command window. For more information, see Accessing Memory by Virtual Address and Accessing Memory by Physical Address.

Opening a Memory Window

To open a Memory window, choose **Memory** from the **View** menu. (You can also press ALT+5 or click the **Memory** button ( ) on the toolbar. ALT+SHIFT+5 closes the active Memory window.)

The following screen shot shows an example of a Memory window.



Using a Memory Window

The Memory window displays data in several columns. The column on the left side of the window shows the beginning address of each line. The remaining columns display the requested information, from left to right. If you select **Bytes** in the **Display format** menu, the ASCII characters that correspond to these bytes are displayed in the right side of the window.

Note By default, the Memory window displays virtual memory. This type of memory is the only type of memory that is available in user mode. In kernel mode, you can use the **Memory Options** dialog box to display physical memory and other data spaces. The **Memory Options** dialog box is described later in this topic.

In the Memory window, you can do the following:

- To write to memory, click inside the Memory window and type new data. You can edit only hexadecimal data—you cannot directly edit ASCII and Unicode characters. Changes take effect as soon as you type new information.

- To see other sections of memory, use the **Previous** and **Next** buttons on the Memory window toolbar, or press the PAGE UP or PAGE DOWN keys. These buttons and keys display the immediately preceding or following sections of memory. If you request an invalid page, an error message appears.
- To navigate within the window, use the RIGHT ARROW, LEFT ARROW, UP ARROW, and DOWN ARROW keys. If you use these keys to move off of the page, a new page is displayed. Before you use these keys, you should resize the Memory window so that it does not have scroll bars. This sizing enables you to distinguish between the actual page edge and the window cutoff.
- To change the memory location that is being viewed, enter a new address into the address box at the top of the Memory window. Note that the Memory window refreshes its display while you enter an address, so you could get error messages before you have completed typing the address. **Note** The address that you enter into the box is interpreted in the current radix. If the current radix is not 16, you should prefix a hexadecimal address with **0x**. To change the default radix, use the **n (Set Number Base)** command in the Debugger Command window. The display within the Memory window itself is not affected by the current radix.
- To change the data type that the window uses to display memory, use the **Display format** menu in the Memory window toolbar. Supported data types include short words, double words, and quad-words; short, long, and quad integers and unsigned integers; 10-byte, 16-byte, 32-byte, and 64-byte real numbers; ASCII characters; Unicode characters; and hexadecimal bytes. The display of hexadecimal bytes includes ASCII characters as well.

The Memory window has a toolbar that contains two buttons, a menu, and a box and has a shortcut menu with additional commands. To access the menu, right-click the title bar or click the icon near the upper-right corner of the window ( ). The toolbar and shortcut menu contain the following choices:

- (Toolbar only) The address box enables you to specify a new address or offset. The exact meaning of this box depends on the memory type you are viewing. For example, if you are viewing virtual memory, the box enables you to specify a new virtual address or offset.
- (Toolbar only) **Display format** enables you to select a new display format.
- (Toolbar and menu) **Previous** (on the toolbar) and **Previous page** (on the shortcut menu) cause the previous section of memory to be displayed.
- (Toolbar and menu) **Next** (on the toolbar) and **Next page** (on the shortcut menu) cause the next section of memory to be displayed.
- (Menu only) **Toolbar** turns the toolbar on and off.
- (Menu only) **Auto-fit columns** ensures that the number of columns displayed in the Memory window fits the width of the Memory window.
- (Menu only) **Dock** or **Undock** causes the window to enter or leave the docked state.
- (Menu only) **Move to new dock** closes the Memory window and opens it in a new dock.
- (Menu only) **Set as tab-dock target for window type** sets the selected Memory window as the tab-dock target for other Memory windows. All Memory windows that are opened after one is chosen as the tab-dock target are automatically grouped with that window in a tabbed collection.
- (Menu only) **Always floating** causes the window to remain undocked even if it is dragged to a docking location.
- (Menu only) **Move with frame** causes the window to move when the WinDbg frame is moved, even if the window is undocked. For more information about docked, tabbed, and floating windows, see Positioning the Windows.

- (Menu only) **Properties** opens the **Memory Options** dialog box, which is described in the following section within this topic.
- (Menu only) **Help** opens this topic in the Debugging Tools for Windows documentation.
- (Menu only) **Close** closes this window.

Memory Options Dialog Box

When you click **Properties** on the shortcut menu, the **Memory Options** dialog box appears.

In kernel mode, there are six memory types available as tabs in this dialog box: **Virtual Memory**, **Physical Memory**, **Bus Data**, **Control Data**, **I/O** (I/O port information), and **MSR** (model-specific register information). Click the tab that corresponds to the information that you want to access.

In user mode, only the **Virtual Memory** tab is available.

Each tab enables you to specify the memory that you want to display:

- In the **Virtual Memory** tab, in the **Offset** box, specify the address or offset of the beginning of the memory range that you want to view.
- In the **Physical Memory** tab, in the **Offset** box, specify the physical address of the beginning of the memory range that you want to view. The Memory window can display only described, cacheable physical memory. If you want to display physical memory that has other attributes, use the **d* (Display Memory)** command or the **!d*** extension.
- In the **Bus Data** tab, in the **Bus Data Type** menu, specify the bus data type. Then, use the **Bus number**, **Slot number**, and **Offset** boxes to specify the bus data that you want to view.
- In the **Control Data** tab, use the **Processor** and **Offset** text boxes to specify the control data that you want to view.
- In the **I/O** tab, in the **Interface Type** menu, specify the I/O interface type. Use the **Bus number**, **Address space**, and **Offset** boxes to specify the data that you want to view.
- In the **MSR** tab, in the **MSR** box, specify the model-specific register that you want to view.

Each tab also includes a **Display format** menu. This menu has the same effect as the **Display format** menu in the Memory window.

Click **OK** in the **Memory Options** dialog box to cause your changes to take effect.

Additional Information

For more information about memory manipulation and a description of other memory-related commands, see Reading and Writing Memory.

Viewing and Editing Global Variables in WinDbg

12/5/2018 • 2 minutes to read • Edit Online

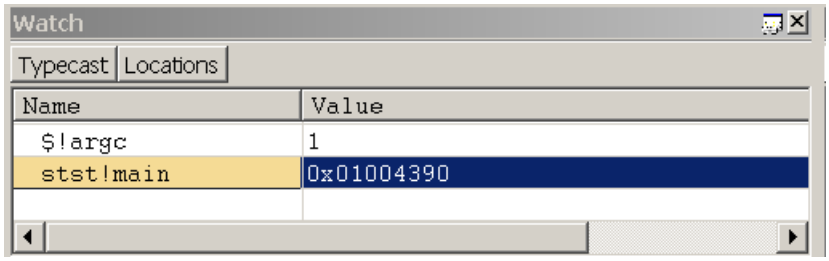
The debugger interprets the name of a global variable as a virtual address. Therefore, you can use all of the commands that are described in Accessing Memory by Virtual Address to read or write global variables.

In addition, you can use the **? (Evaluate Expression)** command to display the address that is associated with any symbol.

In WinDbg, you can also use the Watch window to display and change global and local variables. The Watch window can display any list of variables that you want. These variables can include global variables and local variables from any function. At any time, the Watch window displays the values of those variables that match the current function's scope. You can also change the values of these variables through the Watch window.

To open the Watch window, choose **Watch** from the **View** menu. You can also press ALT+2 or click the **Watch** button on the toolbar: . ALT+SHIFT+2 closes the Watch window.

The following screen shot shows an example of a Watch window.



The Watch window can contain four columns. The **Name** and **Value** columns are always displayed, and the **Type** and **Location** columns are optional. To display the **Type** and **Location** columns, click the **Typecast** and **Locations** buttons, respectively, on the toolbar.

Viewing and Editing Local Variables in WinDbg

12/5/2018 • 7 minutes to read • Edit Online

In WinDbg, you can view local variables by entering commands, by using the Locals window, or by using the Watch window.

Debugger Command Window

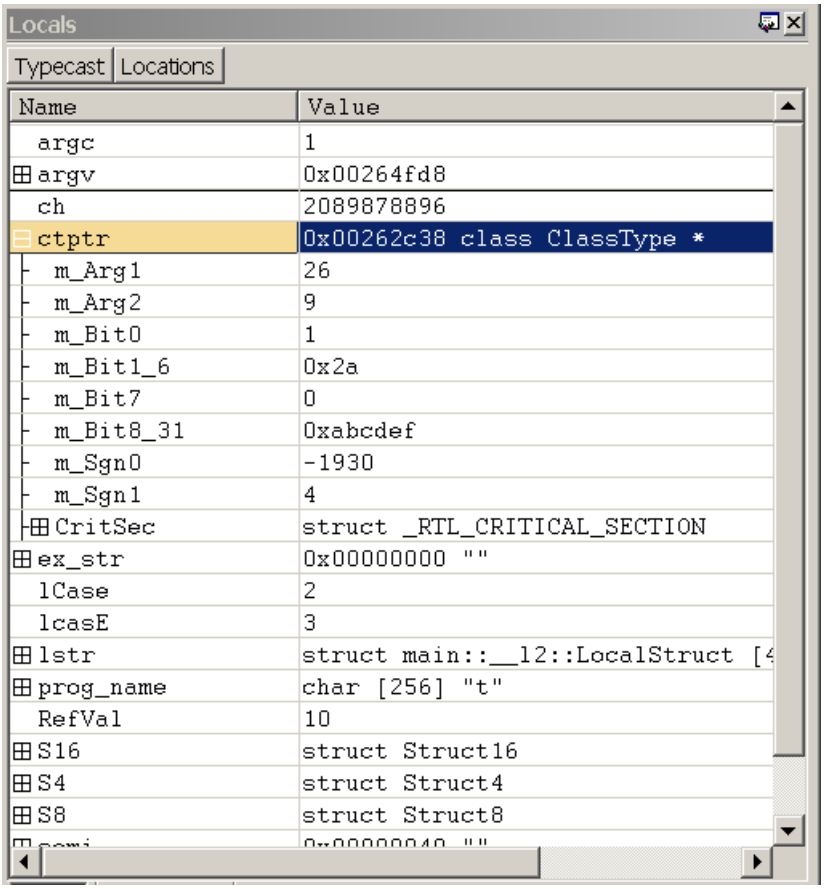
You can view local variables and parameters by entering the **dv** command or the **dt** command in the Debugger Command window.

Opening the Locals Window

The Locals window displays information about all of the local variables in the current scope.

To open or switch to the Locals window, in the WinDbg window, on the **View** menu, click **Locals**. (You can also press ALT+3 or click the **Locals** button ( ) on the toolbar. ALT+SHIFT+3 closes the Locals window.)

The following screen shot shows an example of a Locals window.



The Locals window can contain four columns. The **Name** and **Value** columns are always displayed, and the **Type** and **Location** columns are optional. To display the **Type** and **Location** columns, click the **Typecast** and **Locations** buttons, respectively, on the toolbar.

Using the Locals Window

In the Locals window, you can do the following:

- The **Name** column displays the name of each local variable. If a variable is a data structure, a check box appears next to its name. To expand or collapse the display of structure members, select or clear the check box.
- The **Value** column displays the current value of each variable.
  - To enter a new value for the variable, double-click the current value and type the new value, or edit the old value. (The cut, copy, and paste commands are available to use for editing.) You can type any C++ expression.
  - To save the new value, press ENTER.
  - To discard the new value, press ESC.
  - If you type an invalid value, the old value will reappear when you press ENTER.

  Integers of type **int** are displayed as decimal values; integers of type **UINT** are displayed in the current radix. To change the current radix, use the **n (Set Number Base)** command in the Debugger Command window.
- The **Type** column (if it is displayed in the Locals window) shows the current data type of each variable. Each variable is displayed in the format that is proper for its own data type. Data structures have their type names in the **Type** column. Other variable types display "Enter new type" in this column.

  If you double-click "Enter new type", you can cast the type by entering a new data type. This cast alters the current display of this variable only in the Locals window; it does not change anything in the debugger or on the target computer. Moreover, if you enter a new value in the **Value** column, the text you enter will be parsed based on the actual type of the symbol, rather than any new type you may have entered in the **Type** column. If you close and reopen the Locals window, you will lose the data type changes.

  You can also enter an extension command in the **Type** column. The debugger will pass the address of the symbol to this extension, and will display the resulting output in a series of collapsible rows beneath the current row. For example, if the symbol in this row is a valid address for a thread environment block, you can enter **!teb** in the **Type** column to run the **!teb** extension on this symbol's address.
- The **Location** column (if it is displayed in the Locals window) shows the offset of each member of a data structure.
- If a local variable is an instance of a class that contains Vtables, the **Name** column displays the Vtables, and you can expand the Vtables to show the function pointers. If a Vtable is contained in a base class that points to a derived implementation, the notation **_vtcast_Class** is displayed to indicate the members that are being added because of the derived class. These members expand like the derived class type.
- The local context determines which set of local variables will be displayed in the Locals window. When the local context changes for any reason, the Locals window is automatically updated. By default, the local context matches the current position of the program counter. For more information about how to change the local context, see Local Context.

The Locals window has a toolbar that contains two buttons (**Typecast** and **Locations**) and a shortcut menu with additional commands. To access the menu, right-click the title bar of the window or click the icon near the upper-right corner of the window ( ). The toolbar and menu contain the following buttons and commands:

- (Toolbar and menu) **Typecast** turns the display of the **Type** column on and off.
- (Toolbar and menu) **Locations** turns the display of the **Location** column on and off.
- (Menu only) **Display 16-bit values as Unicode** displays Unicode strings in this window. This command turns on and off a global setting that affects the Locals window, the Watch window, and debugger command output. This command is equivalent to using the **.enable_unicode (Enable Unicode Display)** command.

- (Menu only) **Always display numbers in default radix** causes integers to be displayed in the default radix instead of displaying them in decimal format. This command turns on and off a global setting that affects the Locals window, the Watch window, and debugger command output. This command is equivalent to using the **.force_radix_output (Use Radix for Integers)** command.

  Note The **Always display numbers in default radix** command does not affect long integers. Long integers are displayed in decimal format unless the **.enable_long_status (Enable Long Integer Display)** command is set. The **.enable_long_status** command affects the display in the Locals window, the Watch window, and in debugger command output; there is no equivalent for this command in the menu in the Locals window.

- (Menu only) **Open memory window for selected value** opens a new docked Memory window that displays memory starting at the address of the selected expression.

- (Menu only) **Invoke dt for selected memory value** runs the **dt (Display Type)** command with the selected symbol as its parameter. The result appears in the Debugger Command window. The **-n** option is automatically used to differentiate the symbol from a hexadecimal address. No other options are used. Note that the content that is produced by using this menu selection is identical to the content produced when running the **dt** command from the command line, but the format is slightly different.

- (Menu only) **Toolbar** turns the toolbar on and off.

- (Menu only) **Dock** or **Undock** causes the window to enter or leave the docked state.

- (Menu only) **Move to new dock** closes the Locals window and opens it in a new dock.

- (Menu only) **Set as tab-dock target for window type** is unavailable for the Locals window. This option is only available for Source or Memory windows.

- (Menu only) **Always floating** causes the window to remain undocked even if it is dragged to a docking location.

- (Menu only) **Move with frame** causes the window to move when the WinDbg frame is moved, even if the window is undocked. For more information about docked, tabbed, and floating windows, see Positioning the Windows.

- (Menu only) **Help** opens this topic in the Debugging Tools for Windows documentation.

- (Menu only) **Close** closes this window.

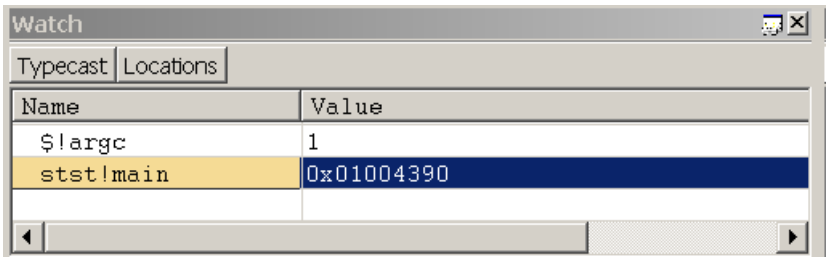
The Watch Window

In WinDbg, you can use the Watch window to display and change local variables. The Watch window can display any list of variables that you want. These variables can include global variables and local variables from any function. At any time, the Watch window displays the values of those variables that match the current function's scope. You can also change the values of these variables through the Watch window.

Unlike the Locals window, the Watch window is not affected by changes to the local context. Only those variables that are defined in the scope of the current program counter can have their values displayed or modified.

To open the Watch window, choose **Watch** from the **View** menu. You can also press ALT+2 or click the **Watch** button on the toolbar: . ALT+SHIFT+2 closes the Watch window.

The following screen shot shows an example of a Watch window.



The Watch window can contain four columns. The **Name** and **Value** columns are always displayed, and the **Type** and **Location** columns are optional. To display the **Type** and **Location** columns, click the **Typecast** and **Locations** buttons, respectively, on the toolbar.

Additional Information

For more information about controlling local variables, an overview of using variables and changing the scope, and a description of other memory-related commands, see Reading and Writing Memory.

Viewing and Editing Registers in WinDbg

12/5/2018 • 4 minutes to read • Edit Online

Registers are small volatile memory units that are located on the CPU. Many registers are dedicated to specific uses, and other registers are available for user-mode applications to use. The x86-based and x64-based processors have different collections of registers available. For more information about the registers on each processor, see Processor Architecture.

In WinDbg, you can view and edit registers by entering commands, by using the Registers window, or by using the Watch Window.

Commands

You can view and edit registers by entering the **r (Registers)** command in the Debugger Command window. You can customize the display by using several options or by using the **rm (Register Mask)** command.

Registers are also automatically displayed every time that the target stops. If you are stepping through your code with the **p (Step)** or **t (Trace)** commands, you see a register display at every step. To stop this display, use the **r** option when you use these commands.

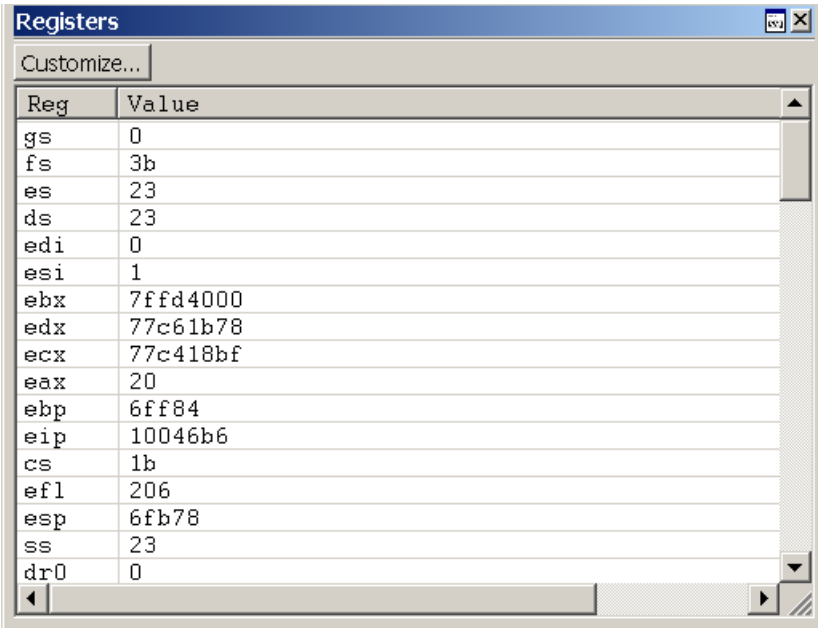
On an x86-based processor, the **r** option also controls several one-bit registers known as flags. To change these flags, you use a slightly different syntax than when changing regular registers. For more information about these flags and an explanation of this syntax, see x86 Flags.

The Registers Window

Opening the Registers Window

To open or switch to the Registers window, choose **Registers** from the **View** menu. (You can also press ALT+4 or click the **Registers** button ( ) on the toolbar. ALT+SHIFT+4 closes the Registers window.)

The following screen shot shows an example of a Registers window.



The Registers window contains two columns. The **Reg** column lists all of the registers for the target processor. The **Value** column displays the current value of each register. This window also contains a **Customize** button on

the toolbar that opens the **Customize Register List** dialog box.

Using the Registers Window

In the Registers window, you can do the following:

- The **Value** column displays the current value of each register. The value of the most recently changed register is displayed in red text.
  - To enter a new value, double-click a **Value** cell, and then type a new value or edit the old value. (The cut, copy, and paste commands are available to use for editing.)
  - To save the new value, press ENTER.
  - To discard the new value, press ESC.
  - If you type an invalid value, the old value will reappear when you press ENTER.
- Register values are displayed in the current radix, and you must type new values in the same radix. To change the current radix, use the **n (Set Number Base)** command in the Debugger Command window.
- In user mode, the Registers window displays the registers that are associated with the current thread. For more information about the current thread, see Controlling Processes and Threads.
- In kernel mode, the Registers window displays the registers that are associated with the current register context. You can set the register context to match a specific thread, context record, or trap frame. Only the most important registers for the specified register context are actually displayed; you cannot change their values.

The Registers window has a toolbar that contains a **Customize** button and has a shortcut menu with additional commands. To access the menu, right-click the title bar or click the icon near the upper-right corner of the window ( ). The toolbar and menu contain the following button and commands:

- (Toolbar and menu) **Customize** opens the **Customize Register List** dialog box, which is described in the following section within this topic.
- (Menu only) **Toolbar** turns the toolbar on and off.
- (Menu only) **Dock** or **Undock** causes the window to enter or leave the docked state.
- (Menu only) **Move to new dock** closes the Registers window and opens it in a new dock.
- (Menu only) **Set as tab-dock target for window type** is unavailable for the Registers window. This option is only available for Source or Memory windows.
- (Menu only) **Always floating** causes the window to remain undocked even if it is dragged to a docking location.
- (Menu only) **Move with frame** causes the window to move when the WinDbg frame is moved, even if the window is undocked. For more information about docked, tabbed, and floating windows, see Positioning the Windows.
- (Menu only) **Help** opens this topic in the Debugging Tools for Windows documentation.
- (Menu only) **Close** closes this window.

Customize Register List Dialog Box

To change the list of registers that are displayed, click the **Customize** button. The **Customize Register List** dialog box will appear.

In this dialog box, you can edit the list of registers to change the order in which registers are displayed. (You cannot actually delete a register from the list; if you do, it will reappear at the end.) There must be a space between register names.

If you select the **Display modified register values first** check box, the register whose values have been changed most recently appears at the top.

If you select the **Do not display subregisters** check box, subregisters are not displayed. For example, **eax** will be displayed, but not **ax**, **ah**, or **al**.

Click **OK** to save your changes or **Cancel** to discard your changes.

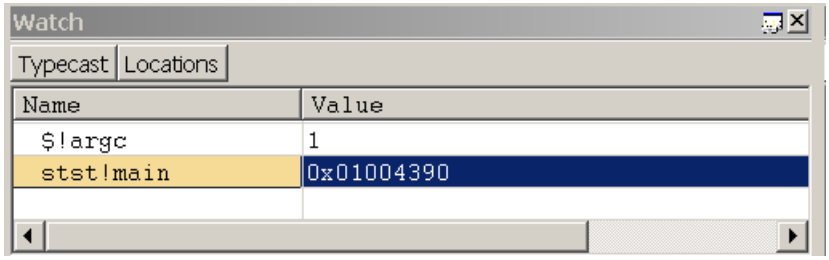
If you are debugging a multi-processor computer with more than one kind of processor, WinDbg stores the customization settings for each processor type separately. This separation enables you to customize the display of each processor's registers simultaneously.

The Watch Window

In WinDbg, you can use the Watch window to display registers. You cannot use the Watch window to alter the values of registers.

To open the Watch window, choose **Watch** from the **View** menu. You can also press ALT+2 or click the **Watch** button on the toolbar: ALT+SHIFT+2 closes the Watch window.

The following screen shot shows an example of a Watch window.



Additional Information

For more information about the register context and other context settings, see Changing Contexts.

Controlling Processes and Threads in WinDbg

12/5/2018 • 2 minutes to read • Edit Online

In WinDbg, the Processes and Threads window displays information about the systems, processes, and threads that are being debugged. This window also enables you to select a new system, process, and thread to be active.

Opening the Processes and Threads Window

To open the Processes and Threads window, choose **Processes and Threads** from the **View** menu. (You can also press ALT+9 or click the **Processes and Threads** button ( ) on the toolbar. ALT+SHIFT+9 closes the Processes and Threads window.)

The following screen shot shows an example of a Processes and Threads window.

The Processes and Threads window displays a list of all processes that are currently being debugged. The threads in the process appear under each process. If the debugger is attached to multiple systems, the systems are shown at the top level of the tree, with the processes subordinate to them, and the threads subordinate to the processes.

Each system listing includes the server name and the protocol details. The system that the debugger is running on is identified as **<Local>**.

Each process listing includes the internal decimal process index that the debugger uses, the hexadecimal process ID, and the name of the application that is associated with the process.

Each thread listing includes the internal decimal thread index that the debugger uses and the hexadecimal thread ID.

Using the Processes and Threads Window

In the Processes and Threads window, the current or active system, process, and thread appear in bold type. To make a new system, process, or thread active, click its line in the window.

The Processes and Threads window has a shortcut menu with additional commands. To access the menu, right-click the title bar or click the icon near the upper-right corner of the window ( ). The following list describes some of the menu commands:

- **Move to new dock** closes the Processes and Threads window and opens it in a new dock.
- **Always floating** causes the window to remain undocked even if it is dragged to a docking location.
- **Move with frame** causes the window to move when the WinDbg frame is moved, even if the window is undocked.

Additional Information

For other methods of displaying or controlling systems, see Debugging Multiple Targets. For other methods of displaying or controlling processes and threads, see Controlling Processes and Threads.

Configuring Exceptions and Events in WinDbg

12/5/2018 • 2 minutes to read • Edit Online

You can configure WinDbg to react to specified exceptions and events in a specific way. For each exception, you can set the break status and the handling status. For each event, you can set the break status.

You can configure the break status by doing one of the following:

- Choose **Event Filters** from the **Debug** menu, click the event that you want from the list in the **Event Filters** dialog box, and then select **Enabled**, **Disabled**, **Output**, or **Ignore**.
- Use the **SXE**, **SXD**, **SXN**, or **SXI** command.

You can configure the handling status by doing one of the following:

- Choose **Event Filters** from the **Debug** menu, click the event that you want from the list in the **Event Filters** dialog box, and then select **Handled** or **Not Handled**.
- Use the **SXE**, **SXD**, **SXN**, or **SXI** command.

For a detailed discussion of exceptions and events, see Controlling Exceptions and Events.

Keeping a Log File in WinDbg

12/5/2018 • 2 minutes to read • Edit Online

WinDbg can write a log file that records the debugging session. This log file contains all of the contents of the Debugger Command window, including the commands that you type and the responses from the debugger.

Opening a New Log File

To open a new log file, or to overwrite a previous log file, do one of the following:

- Choose **Open/Close Log file** from the **Edit** menu.
- When you start WinDbg in a Command Prompt window, use the **-logo** command-line option.
- Enter the **.logopen (Open Log File)** command. If you use the **/t** option, the date and time are appended to your specified file name. If you use the **/u** option, the log file is written in Unicode instead of in ASCII.

Appending to an Existing Log File

To append command window text to a log file, do one of the following:

- Choose **Open/Close Log file** from the **Edit** menu.
- When you start WinDbg in a Command Prompt window, use the **-loga** command-line option.
- Enter the **.logappend (Append Log File)** command. If you are appending to a Unicode log file, you must use the **/u** option.

Closing a Log File

To close an open log file, do one of the following:

- Choose **Open/Close Log file** from the **Edit** menu.
- Enter the **.logclose (Close Log File)** command.

Checking Log File Status

To determine the log file status, do one of the following:

- Choose **Open/Close Log file** from the **Edit** menu, and see whether a log file is listed.
- Enter the **.logfile (Display Log File Status)** command.

Using the Watch Window

12/5/2018 • 7 minutes to read • Edit Online

The Watch window displays information about global variables, local variables, and registers. You can customize this window to show the items that you are tracking.

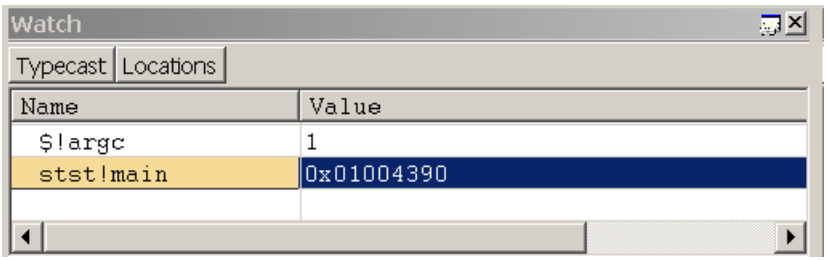
Opening the Watch Window

To open or switch to the Watch window, in the WinDbg window, on the **View** menu, click **Watch**.

You can also press ALT+2 or click the **Watch (ALT+2)** button on the toolbar:

ALT+SHIFT+2 will close the Watch window.

The following screen shot shows an example of a Watch window.



The Watch window can contain four columns. The **Name** and **Value** columns are always displayed, and the **Type** and **Location** columns are optional. To display the **Type** and **Location** columns, click the **Typecast** and **Locations** buttons, respectively, on the toolbar.

Using the Watch Window

In the Watch window, you can do the following:

- To add a variable to the Watch window, select the first empty cell in the **Name** column, type the variable name, and then press ENTER. Separate the module name from the variable with an exclamation point (**!**). If you do not specify a module, the current module is used. To enter an address in the **Name** field, the address must begin with a decimal digit (if necessary, use the prefix **0x**).

  If the variable name that you have entered is defined in the current function's scope, its value appears in the **Value** column. If it is not defined, the **Value**column displays "Error: Cannot get value".

  Even if a variable is not defined, it can be useful to add it to the Watch window. If the program counter enters a function in which a variable of this name is defined, its value appears in the window at that time.

- To remove a variable from the Watch window, double-click its name, press DELETE, and then press ENTER. You can also replace an old name with a new name by double-clicking the old name, typing the new name, and then pressing ENTER.

- If a variable is a data structure, a check box appears next to its name. To expand and collapse the display of structure members, select or clear the check box.

- Integers of type **int** are displayed as decimal values; integers of type **UINT** are displayed in the current radix. To change the current radix, use the **n (Set Number Base)** command in the Debugger Command window.

- To change the value of a local variable, double-click its **Value** cell. Enter the new value, or edit the old value. (The cut, copy, and paste commands are available to use for editing.) The value that you enter can include

any C++ expression. After you enter a new value or edit the old value, you can press ENTER to store the new value or press ESC to discard it. If you submit an invalid value, the old value will reappear after you press ENTER.

Integers of type **int** are displayed as decimal values; integers of type **UINT** are displayed in the current radix. To change the current radix, use the **n (Set Number Base)** command in the Debugger Command window.

- The **Type** column (if it is displayed in the Watch window) shows the current data type of each variable. Each variable is displayed in the format that is proper for its own data type. Data structures have their type names in the **Type** column. Other variable types display "Enter new type" in this column.

  If you double-click "Enter new type", you can cast the type by entering a new data type. This cast alters the current display of this variable only in the Watch window; it does not change anything in the debugger or on the target computer. Moreover, if you enter a new value in the **Value** column, the text you enter will be parsed based on the actual type of the symbol, rather than any new type you may have entered in the **Type** column. If you close and reopen the Watch window, you will lose the data type changes.

  You can also enter an extension command in the **Type** column. The debugger will pass the address of the symbol to this extension, and will display the resulting output in a series of collapsible rows beneath the current row. For example, if the symbol in this row is a valid address for a thread environment block, you can enter **!teb** in the **Type** column to run the **!teb** extension on this symbol's address.

- The **Location** column (if it is displayed in the Watch window) shows the offset of each member of a data structure.

- In addition to variables, you can also monitor the following items in the Watch window:

  - Registers. When you add a register to the Watch window, prefix its name with an at sign (**@**). Unlike variables, you cannot change register values through the Watch window.
  - Vtables that contain function pointers. When a Vtable appears in the Watch window, you can browse the function entries in the table. If a Vtable is contained in a base class that points to a derived implementation, the notation _**vtcast**_*Class* is displayed to indicate the members that are being added because of the derived class. These members expand like the derived class type.
  - The return values of extension functions, such as _EFN_GetPoolData.

Unlike the Locals window, the Watch window is not affected by changes to the register context. In the Watch window, you can see and modify only those variables that are defined in the scope of the current program counter.

If you open a new workspace, the Watch window contents are discarded and replaced with those in the new workspace.

Toolbar and Shortcut Menu

The Watch window has a toolbar that contains two buttons (**Typecast** and **Locations**) and a shortcut menu with additional commands. To access the menu, right-click the title bar of the window or click the icon near the upper-right corner of the window:

The toolbar and menu contain the following buttons and commands:

- (Toolbar and menu) **Typecast** turns the display of the **Type** column on and off.

- (Toolbar and menu) **Locations** turns the display of the **Location** column on and off.

- (Menu only) **Display 16-bit values as Unicode** displays Unicode strings in this window. This command turns on and off a global setting that affects the Locals window, the Watch window, and debugger command output. This command is equivalent to using the **.enable_unicode (Enable Unicode Display)** command.

- (Menu only) **Always display numbers in default radix** causes integers to be displayed in the default radix

instead of always displaying them in decimal format. This command turns on and off a global setting that affects the Locals window, the Watch window, and debugger command output. This command is equivalent to using the **.force_radix_output (Use Radix for Integers)** command. **Note** The **Always display numbers in default radix** command does not affect long integers. Long integers are displayed in decimal format unless the **.enable_long_status (Enable Long Integer Display)** command is used. The **.enable_long_status** command affects the display in the Locals window, the Watch window, and debugger command output. There is no equivalent for this command in the menu in the Watch window.

- (Menu only) **Open memory window for selected value** opens a new docked Memory window that displays memory starting at the address of the selected expression.
- (Menu only) **Invoke dt for selected memory value** runs the **dt (Display Type)** command with the selected symbol as its parameter. The result appears in the Debugger Command window. The **-n** option is automatically used to differentiate the symbol from a hexadecimal address. No other options are used. Note that the content produced using this menu selection is identical to the content produced when running the **dt** command from the command line, but the format is slightly different.
- (Menu only) **Toolbar** turns the toolbar on and off.
- (Menu only) **Dock** or **Undock** causes the window to enter or leave the docked state.
- (Menu only) **Move to new dock** closes the Watch window and opens it in a new dock.
- (Menu only) **Set as tab-dock target for window type** is unavailable for the Watch window. This option is only available for Source or Memory windows.
- (Menu only) **Always floating** causes the window to remain undocked even if it is dragged to a docking location.
- (Menu only) **Move with frame** causes the window to move when the WinDbg frame is moved, even if the window is undocked. For more information about docked, tabbed, and floating windows, see Positioning the Windows.
- (Menu only) **Help** opens this topic in the Debugging Tools for Windows documentation.
- (Menu only) **Close** closes this window.

Additional Information

For more information about controlling variables and a description of other memory-related commands, see Reading and Writing Memory. For more information about registers and their manipulation, see Viewing and Editing Registers in WinDbg. For more information about docked, tabbed, and floating windows, see Positioning the Windows. For more information about all techniques that you can use to control debugging information windows, see Using Debugging Information Windows.

Using the Scratch Pad

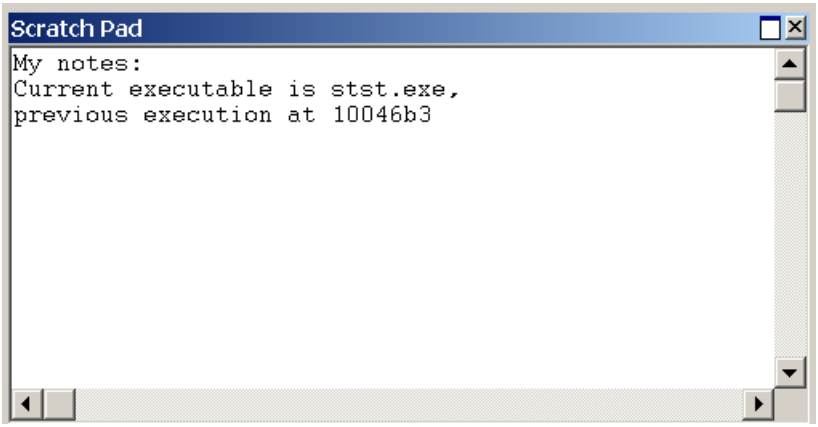
12/5/2018 • 2 minutes to read • Edit Online

The Scratch Pad window is a clipboard on which you can type and save text.

Opening the Scratch Pad Window

To open or switch to the Scratch Pad window, in the WinDbg window, on the **View** menu, click **Scratch Pad**. (You can also press ALT+8 or click the **Scratch Pad (Alt+8)** button ( ) on the toolbar.)

The following screen shot shows an example of a Scratch Pad window.



Using the Scratch Pad Window

In the Scratch Pad window, you can do the following:

- To type in the Scratch Pad window, click in the window where you want to add text and begin typing. You can also use standard copy-and-paste features. The contents of the Scratch Pad window do not affect the operation of the debugger. This window exists solely to help with text editing.
- If you close the Scratch Pad window, your text is preserved and is available when you reopen the window. You can also save text from the Scratch Pad window by associating it with a file.

The Scratch Pad window has a shortcut menu with additional commands. To access the menu, right-click the title bar or click the icon near the upper-right corner of the window ( ). This menu contains the following commands:

- (Menu only) **Associate with file** opens a dialog box through which you can select a text file. After the file is selected, the current text in the Scratch Pad is cleared and replaced with the text in the selected file. While Scratch Pad is associated with this file, all new text typed into Scratch Pad is saved to the file. Association with the file can be ended either by selecting the **End file association** short-cut menu option or by closing and reopening Scratch Pad.
- (Menu only) **End file association** ends Scratch Pad's association with a specified text file. All text in Scratch Pad prior to selecting this option is saved in the file. All text typed in Scratch Pad after the association is ended is no longer saved in the text file.
- **Dock** or **Undock** causes the window to enter or leave the docked state.
- (Menu only) **Move to new dock** closes Scratch Pad and opens it in a new dock.
- (Menu only) **Set as tab-dock target for window type** is unavailable for Scratch Pad. This option is only available for Source or Memory windows.
- **Always floating** causes the window to remain undocked even if it is dragged to a docking location.

- **Move with frame** causes the window to move when the WinDbg frame is moved, even if the window is undocked. For more information about docked, tabbed, and floating windows, see Positioning the Windows.
- **Help** opens this topic in the Debugging Tools for Windows documentation.
- **Close** closes this window.

Additional Information

For more information about docked, tabbed, and floating windows, see Positioning the Windows. For more information about all techniques that you can use to control debugging information windows, see Using Debugging Information Windows.

Debugging Using KD and NTKD

12/5/2018 • 2 minutes to read • Edit Online

This section describes how to perform basic debugging tasks using the KD and NTKD debuggers.

KD and NTKD are identical in every way, except that NTKD spawns a new text window when it is started, whereas KD inherits the Command Prompt window from which it was invoked. The instructions in this section are given for KD, but they work equally well for NTKD. For a discussion of when to use KD or NTKD, see Debugging Environments.

Details are given in the following topics:

- Opening a Dump File Using KD
- Live Kernel-Mode Debugging Using KD
- Ending a Debugging Session in KD
- Setting Symbol and Executable Image Paths in KD
- Setting Breakpoints in KD
- Viewing the Call Stack in KD
- Viewing and Editing Memory in KD
- Viewing and Editing Registers in KD
- Remote Debugging Using KD
- Configuring Exceptions and Events in KD
- Keeping a Log File in KD

Opening a Dump File Using KD

12/5/2018 • 2 minutes to read • Edit Online

Command Prompt

In a Command Prompt window, you can open a dump file when you launch KD. Use the following command.

kd -y *SymbolPath* **-i** *ImagePath* **-z** *DumpFileName*

The **-v** option (verbose mode) is also useful. For more information about the command-line syntax, see **KD Command-Line Options**.

KD Command Line

You can also open a dump file after the debugger is running by entering the **.opendump (Open Dump File)** command, followed by **g (Go)**.

Live Kernel-Mode Debugging Using KD

12/5/2018 • 4 minutes to read • Edit Online

In a Command Prompt window, you can initiate a live kernel-mode debugging session when you launch KD. Enter one of the following commands.

kd [-y *SymbolPath*] -k net:port=*PortNumber*,key=*Key*[,target=*TargetIPAddress*|*TargetHostName*]

kd [-y *SymbolPath*] -k 1394:channel=*1394Channel*[,symlink=*1394Protocol*]

kd [-y *SymbolPath*] -k usb:targetname=*USBString*

kd [-y *SymbolPath*] -k com:port=*ComPort*,baud=*BaudRate*

kd [-y *SymbolPath*] -k com:pipe,port=*VMHost*\pipe*PipeName*[,resets=0][,reconnect]

kd [-y *SymbolPath*] -k com:*modem*

kd [-y *SymbolPath*] -kl

kd [-y *SymbolPath*] -k

For more information, see **KD Command-Line Options**.

Environment Variables

For debugging over a serial (COM port) or 1394 connection, you can use environment variables to specify the connection settings.

Use the following variables to specify a serial connection.

set _NT_DEBUG_PORT = *ComPort*

set _NT_DEBUG_BAUD_RATE = *BaudRate*

Use the following variables to specify a 1394 connection.

set _NT_DEBUG_BUS = 1394

set _NT_DEBUG_1394_CHANNEL = *1394Channel*

set _NT_DEBUG_1394_SYMLINK = *1394Protocol*

For more information, see Kernel-Mode Environment Variables.

Parameters

SymbolPath
A list of directories where symbol files are located. Directories in the list are separated by semicolons. For more information, see Symbol Path.

PortNumber
A port number to use for network debugging. You can choose any number from 49152 through 65535. For more information, see Setting Up a Network Connection Manually.

Key
The encryption key to use for network debugging. We recommend that you use an automatically generated key, which is provided by bcdedit when you configure the target computer. For more information, see Setting Up a Network Connection Manually.

TargetIPAddress
The IPv4 address of the target machine.

When the target= IP address is specified, this causes the debugger to initiate a connection to the specified target machine, by sending a special packet to the target, that will cause it to attempt to connect with that debugger. The debugger will send packets to the target repeatedly approximately every half second, attempting to connect. If the connection is successful, the target will drop any existing connection, and communicate only with this instance of the debugger. This allows you to take control of the debugging session away from an existing debugging connection.

When the target is configured with a host IP address, and the debugger is being run on the machine with the configured host IP address, there is no need to specify the target= IP address parameter. When the target is configured with a host IP address, it will send OFFER packets to the host every three seconds. The OFFER packets allow the debugger to connect to the host when no target= IP address is specified.

For more information on configuring the host IP address on the target, see Setting Up KDNET Network Kernel Debugging Automatically and Setting Up KDNET Network Kernel Debugging Manually.

TargetMachineName
The machine name of the target PC. To use the machine name, the DNS system on the network must have the machine name associated with the IP address of the target PC.

1394Channel
The 1394 channel number. Valid channel numbers are any integer between 0 and 62, inclusive. *1394Channel* must match the number used by the target computer, but does not depend on the physical 1394 port chosen on the adapter. For more information, see Setting Up a 1394 Connection Manually.

1394Protocol
The connection protocol to be used for the 1394 kernel connection. This can almost always be omitted, because the debugger will automatically choose the correct protocol. If you wish to set this manually, and the target computer is running Windows XP, *1394Protocol* should be set equal to "channel". If the target computer is running Windows Server 2003 or later, *1394Protocol* should be set equal to "instance". If it is omitted, the debugger will default to the protocol appropriate for the current target computer. This can only be specified through the command line or the environment variables, not through the WinDbg graphical interface.

USBString
A USB connection string. This must match the string specified with the /targetname boot option. For more information, see Setting Up a USB 3.0 Connection Manually and Setting Up a USB 2.0 Connection Manually.

ComPort
The name of the COM port. This can be in the format "com2" or in the format "\\.\com2", but should not simply be a number. For more information, see Setting Up a Serial Connection Manually.

BaudRate
The baud rate. This can be 9600, 19200, 38400, 57600, or 115200.

VMHost
When debugging a virtual machine, *VMHost* specifies the name of the physical computer on which the virtual machine is running. If the virtual machine is running on the same computer as the kernel debugger itself, use a single period (.) for *VMHost*. For more information, see Setting Up a Connection to a Virtual Machine.

PipeName
The name of the pipe created by the virtual machine for the debugging connection.

resets=0
Specifies that an unlimited number of reset packets can be sent to the target when the host and target are synchronizing. This parameter is only needed when debugging certain kinds of virtual machines.

reconnect
Causes the debugger to automatically disconnect and reconnect the pipe if a read/write failure occurs. Additionally, if the named pipe is not found when the debugger is started, the reconnect parameter will cause it to wait for a pipe of this name to appear. This parameter is only needed when debugging certain kinds of virtual machines.

-kl
Causes the debugger to perform local kernel-mode debugging. For more information, see Local Kernel-Mode Debugging.

Examples

The following batch file could be used to set up and start a debugging session over a COM port connection.

```
set _NT_SYMBOL_PATH=d:\mysymbols
set _NT_DEBUG_PORT=com1
set _NT_DEBUG_BAUD_RATE=115200
set _NT_DEBUG_LOG_FILE_OPEN=d:\debuggers\logfile1.log
kd
```

The following batch file could be used to set up and start a debugging session over a 1394 connection.

```
set _NT_SYMBOL_PATH=d:\mysymbols
set _NT_DEBUG_BUS=1394
set _NT_DEBUG_1394_CHANNEL=44
set _NT_DEBUG_LOG_FILE_OPEN=d:\debuggers\logfile1.log
kd
```

The following command lines could be used to start WinDbg without any environment variables.

kd -y d:\mysymbols -k com:port=com2,baud=57600

kd -y d:\mysymbols -k com:port=\\.\com2,baud=115200

kd -y d:\mysymbols -k 1394:channel=20,symlink=instance

kd -y d:\mysymbols -k net:port=50000,key=*AutoGeneratedKey*

Ending a Debugging Session in KD

12/5/2018 • 2 minutes to read • Edit Online

There are two different ways to exit KD.

- Issue a **q (Quit)** command in KD to save the log file, end the debugging session, and exit the debugger. The target computer remains locked.
- Press **CTRL+B** and then press ENTER to end the debugger abruptly. If you want the target computer to continue to run after the debugger is ended, you must use this method. Because CTRL+B does not remove breakpoints, you should use the following commands first.

```
kd>  bc *
kd>  g
```

Exiting the debugger by using CTRL+B does not clear kernel-mode breakpoints, but attaching a new kernel debugger does clear these breakpoints.

When you are performing remote debugging, **q** ends the debugging session. CTRL+B exits the debugger but leaves the session active. This situation enables another debugger to connect to the session.

If the **q (Quit)** command does not work, press **CTRL+R** and then press ENTER on the host computer's keyboard, and then retry the **q** command. If this procedure does not work, you must use CTRL+B to exit the debugger.

Setting Symbol and Executable Image Paths in KD

12/5/2018 • 2 minutes to read • Edit Online

Symbol Path

The symbol path specifies the directories where the symbol files are located. For more information about symbols and symbol files, see Symbols.

Note If you are connected to the Internet or a corporate network, the most efficient way to access symbols is to use a symbol server. You can use a symbol server by using the srv* or symsrv* string within your symbol path. For more information about symbol servers, see Symbol Stores and Symbol Servers.

To control the symbol path in KD, do one of the following:

- Enter the **.sympath (Set Symbol Path)** command. If you are using a symbol server, the **.symfix (Set Symbol Store Path)** command is similar to **.sympath** but saves you typing.
- When you start the debugger, use the **-y** command-line option. See **KD Command-Line Options**.
- Before you start the debugger, use the _NT_SYMBOL_PATH and _NT_ALT_SYMBOL_PATH environment variables to set the path. The symbol path is created by appending _NT_SYMBOL_PATH after _NT_ALT_SYMBOL_PATH. (Typically, the path is set through the _NT_SYMBOL_PATH. However, you might want to use _NT_ALT_SYMBOL_PATH to override these settings in special cases, such as when you have private versions of shared symbol files.)

  Note If you use the **-sins** command-line option, the debugger ignores the symbol path environment variable.

Executable Image Path

An executable file is a binary file that the processor can run. These files typically have the .exe, .dll, or .sys file name extension. Executable files are also known as modules, especially when executable files are described as units of a larger application. Before the Windows operating system runs an executable file, it loads it into memory. The copy of the executable file in memory is called the executable image or the image.

Note These terms are sometimes used imprecisely. For example, some documents might use "image" for the actual file on the disk. Also, Windows-based applications refer to the executable name, which typically includes the file name extension. But these applications refer to the module name, which does not include the file name extension. Also, the Windows kernel and HAL have special module names. For example, the **nt** module corresponds to the Ntoskrnl.exe file.

The executable image path specifies the directories that the binary executable files are located in.

In most situations, the debugger knows the location of the executable files, so you do not have to set the path for this file.

However, there are situations when this path is required. For example, kernel-mode small memory dump files do not contain all of the executable files that exist in memory at the time of a stop error (that is, a crash). Similarly, user-mode minidump files do not contain the application binaries. If you set the path of the executable files, the debugger can find these binary files.

The debugger's executable image path is a string that consists of multiple directory paths, separated by semicolons. Relative paths are supported. However, unless you always start the debugger from the same directory,

you should add a drive letter or a network share before each path. Network shares are also supported. The debugger searches the executable image path recursively. That is, the debugger searches the subdirectories of each directory that is listed in this path.

To control the executable image path in KD, do one of the following:

- Enter the **.exepath (Set Executable Path)** command.
- When you start the debugger, use the **-i** command-line option. See **KD Command-Line Options**.
- Before you start the debugger, use the _NT_EXECUTABLE_IMAGE_PATH environment variable to set the path.

  Note If you use the **-sins** command-line option, the debugger ignores the executable image path environment variable.

Setting Breakpoints in KD

12/5/2018 • 2 minutes to read • Edit Online

You can set, view, and manipulate breakpoints in KD by entering commands. For a list of commands, see Methods of Controlling Breakpoints.

Viewing the Call Stack in KD

12/5/2018 • 2 minutes to read • Edit Online

The call stack is the chain of function calls that have led to the current location of the program counter. The top function on the call stack is the current function, the next function is the function that called the current function, and so on. The call stack that is displayed is based on the current program counter, unless you change the register context. For more information about how to change the register context, see Changing Contexts.

In KD, you can view the call stack by entering one of the **k (Display Stack Backtrace)** commands.

Viewing and Editing Memory in KD

12/5/2018 • 2 minutes to read • Edit Online

Viewing and Editing Memory

In KD, you can view and edit memory by entering one of the **Display Memory** commands, and you can edit memory by entering one of the **Enter Values** commands. For a detailed discussion of these commands, see Accessing Memory by Virtual Address and Accessing Memory by Physical Address.

Viewing and Editing Variables

In KD, you can view and edit global variables by entering commands. The debugger interprets the name of a global variable as a virtual address. Therefore, all of the commands that are described in Accessing Memory by Virtual Address can be used to read or write global variables. For additional information about viewing and editing global variables, see Accessing Global Variables.

In KD you can view and edit local variables by entering commands. The debugger interprets the name of a local variable as an address. Therefore, all of the commands that are described in Accessing Memory by Virtual Address can be used to read or write local variables. However, if it is necessary to indicate to a command that a symbol is local, precede the symbol with a dollar sign ($) and an exclamation point (!), as in `$!var`. For additional information about viewing and editing local variables, see Accessing Local Variables.

Viewing and Editing Registers in KD

12/5/2018 • 2 minutes to read • Edit Online

Registers are small volatile memory units that are located on the CPU. Many registers are dedicated to specific uses, and other registers are available for user-mode applications to use. The x86-based and x64-based processors have different collections of registers available. For more information about the registers on each processor, see Processor Architecture.

In KD, you can view and edit registers by entering the **r (Registers)** command. You can customize the display by using several options or by using the **rm (Register Mask)** command.

Registers are also automatically displayed every time that the target stops. If you are stepping through your code with the **p (Step)** or **t (Trace)** commands, you see a register display at every step. To stop this display, use the **r** option when you use these commands.

On an x86-based processor, the **r** option also controls several one-bit registers known as flags. To change these flags, you use a slightly different syntax than when changing regular registers. For more information about these flags and an explanation of this syntax, see x86 Flags.

Remote Debugging Using KD

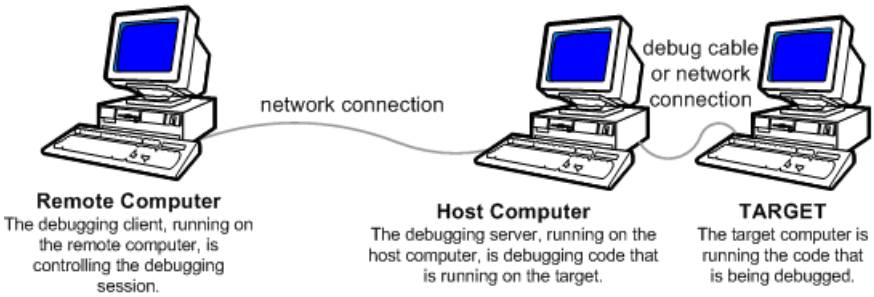
12/5/2018 • 2 minutes to read • Edit Online

Remote debugging involves two debuggers running at two different locations. The debugger that performs the debugging is called the *debugging server*. The second debugger, called the *debugging client*, controls the debugging session from a remote location. To establish a remote session, you must set up the debugging server first and then activate the debugging client.

Remote debugging can be useful when you would like to involve someone else in looking at an issue, that you are debugging on a PC.

The code that is being debugged could be running on the same computer that is running the debugging server, or it could be running on a separate computer. If the debugging server is performing user-mode debugging, then the process that is being debugged can run on the same computer as the debugging server. If the debugging server is performing kernel-mode debugging, then the code being debugged would typically run on a separate target computer.

The following diagram illustrates a remote session where the debugging server, running on a host computer, is performing kernel-mode debugging of code that is running on a separate target computer.



There are several transport protocols you can use for a remote debugging connection: TCP, NPIPE, SPIPE, SSL, and COM Port. Suppose you have chosen to use TCP as the protocol and you have chosen to use KD as both the debugging client and the debugging server. You can use the following procedure to establish a remote kernel-mode debugging session:

1. On the host computer, open KD and establish a kernel-mode debugging session with a target computer. (See Performing Kernel-Mode Debugging Using KD.)
2. Break in by pressing CRTL-Break.
3. Enter the following command.

   .server tcp:port=5005

   Note The port number 5005 is arbitrary. The port number is your choice.
4. KD will respond with output similar to the following.

```
Server started.  Client can connect with any of these command lines
0: <debugger> -remote tcp:Port=5005,Server=YourHostComputer
```

5. On the remote computer, open a Command Prompt window, and enter the following command.

   kd -remote tcp:Port=5005,Server=*YourHostComputer*

where *YourHostComputer* is the name of your host computer, which is running the debugging server.

Additional Information

For complete information about launching KD (and establishing remote debugging) at the command line, see **KD Command-Line Options**.

Configuring Exceptions and Events in KD

12/5/2018 • 2 minutes to read • Edit Online

You can configure KD to react to specified exceptions and events in a specific way. For each exception, you can set the break status and the handling status. For each event, you can set the break status.

You can configure the break status or the handling status by using the **SXE**, **SXD**, **SXN**, or **SXI** command.

For a detailed discussion of exceptions and events, see Controlling Exceptions and Events.

Keeping a Log File in KD

12/5/2018 • 2 minutes to read • Edit Online

KD can write a log file that records the debugging session. This log file contains all of the commands that you type and the responses from the debugger.

Opening a New Log File

To open a new log file, or to overwrite a previous log file, do one of the following:

- Before you start the debugger, set the _NT_DEBUG_LOG_FILE_OPEN environment variable.
- When you start the debugger, use the **-logo** command-line option.
- Enter the **.logopen (Open Log File)** command. If you use the **/t** option, the date and time are appended to your specified file name. If you use the **/u** option, the log file is written in Unicode instead of in ASCII.

Appending to an Existing Log File

To append text to an existing log file, do one of the following:

- Before you start the debugger, set the _NT_DEBUG_LOG_FILE_APPEND environment variable.
- When you start the debugger, use the **-loga** command-line option.
- Enter the **.logappend (Append Log File)** command. If you are appending to a Unicode log file, you must use the **/u** option.

Closing a Log File

To close an open log file, do the following:

- Enter the **.logclose (Close Log File)** command.

Checking Log File Status

To determine the log file status, do the following:

- Enter the **.logfile (Display Log File Status)** command.

Debugging Using CDB and NTSD

12/5/2018 • 2 minutes to read • Edit Online

This section describes how to perform basic debugging tasks using the Microsoft Console Debugger (CDB) and Microsoft NT Symbolic Debugger (NTSD).

CDB and NTSD are identical in every way, except that NTSD spawns a new text window when it is started, whereas CDB inherits the Command Prompt window from which it was invoked. The instructions in this section are given for CDB, but they work equally well for NTSD. For a discussion of when to use CDB or NTSD, see Debugging Environments.

Details are given in the following topics:

- Debugging a User-Mode Process Using CDB
- Opening a Dump File Using CDB
- Ending a Debugging Session in CDB
- Setting Symbol and Executable Image Paths in CDB
- Setting Breakpoints in CDB
- Viewing the Call Stack in CDB
- Viewing and Editing Memory in CDB
- Viewing and Editing Registers in CDB
- Configuring Exceptions and Events in CDB
- Keeping a Log File in CDB

Debugging a User-Mode Process Using CDB

12/5/2018 • 3 minutes to read • Edit Online

You can use CDB to attach to a running process or to spawn and attach to new process.

Attaching to a Running Process

Command Prompt

In a Command Prompt window, you can attach to a running process when you launch CDB. Use one of the following commands:

- **cdb -p** *ProcessID*
- **cdb -pn** *ProcessName*

where *ProcessID* is the Process ID of a running process or *ProcessName* is the name of a running process.

For more information about the command-line syntax, see **CDB Command-Line Options**.

CDB Command Window

If the debugger is already debugging one or more processes, you can attach to a running process by using the **.attach (Attach to Process)** command.

The debugger always starts multiple target processes simultaneously, unless some of their threads are frozen or suspended.

If the **.attach** command is successful, the debugger attaches to the specified process the next time that the debugger issues an execution command. If you use this command several times in a row, execution has to be requested by the debugger as many times as you use this command.

Attaching to a Running Process Noninvasively

If you want to debug a running process and interfere only minimally in its execution, you should debug the process noninvasively.

Command Prompt

To noninvasively debug a running process from the CDB command line, specify the **-pv** option, the **-p** option, and the process ID, in the following syntax.

cdb -pv -p *ProcessID*

Or, to noninvasively debug a running process by specifying the process name, use the following syntax instead.

cdb -pv -pn *ProcessName*

There are several other useful command-line options. For more information about the command-line syntax, see **CDB Command-Line Options**.

CDB Command Window

If the debugger is already active, you can noninvasively debug a running process by entering the **.attach -v (Attach to Process)** command.

You can use the **.attach** command if the debugger is already debugging one or more processes invasively.

If the **.attach -v** command is successful, the debugger debugs the specified process the next time that the

debugger issues an execution command. Because execution is not permitted during noninvasive debugging, the debugger cannot noninvasively debug more than one process at a time. This restriction also means that using the **.attach -v** command might make an existing invasive debugging session less useful.

Spawning a New Process

CDB can start a user-mode application and then debug the application. The application is specified by name. The debugger can also automatically attach to child processes (additional processes that the original target process started).

Processes that the debugger creates (also known as spawned processes) behave slightly differently than processes that the debugger does not create.

Instead of using the standard heap API, processes that the debugger creates use a special debug heap. You can force a spawned process to use the standard heap instead of the debug heap by using the _NO_DEBUG_HEAP environment variable or the **-hd** command-line option.

Also, because the target application is a child process of the debugger, it inherits the debugger's permissions. This permission might enable the target application to perform certain actions that it could not perform otherwise. For example, the target application might be able to affect protected processes.

In a Command Prompt window, you can spawn a new process when you launch CDB. Enter the following command.

cdb [-o] *ProgramName* **[***Arguments***]**

The **-o** option causes the debugger to attach to child processes. There are several other useful command-line options. For more information about the command-line syntax, see **CDB Command-Line Options**.

If the debugger is already debugging one or more processes, you can create a new process by entering the **.create (Create Process)** command.

The debugger will always start multiple target processes simultaneously, unless some of their threads are frozen or suspended.

If the **.create** command is successful, the debugger creates the specified process the next time that the debugger issues an execution command. If you use this command several times in a row, execution has to be requested by the debugger as many times as you use this command.

You can control the application's starting directory by using the **.createdir (Set Created Process Directory)** command before **.create**. You can use the **.createdir -I** command or the **-noinh** command-line option to control whether the target application inherits the debugger's handles.

You can activate or deactivate the debugging of child processes by using the **.childdbg (Debug Child Processes)** command.

Reattaching to a Process

If the debugger stops responding or freezes, you can attach a new debugger to the target process. For more information about how to attach a debugger in this situation, see Reattaching to the Target Application.

Opening a Dump File Using CDB

12/5/2018 • 2 minutes to read • Edit Online

Command Prompt

In a Command Prompt window, you can open a user-mode dump file when you launch CDB. Enter the following command.

cdb -y *SymbolPath* **-i** *ImagePath* **-z** *DumpFileName*

The **-v** option (verbose mode) is also useful. For more information about the command-line syntax, see **CDB Command-Line Options**

CDB Command Line

You can also open a dump file after the debugger is running by entering the **.opendump (Open Dump File)** command, followed by **g (Go)**. This allows you to debug multiple dump files at the same time.

Ending a Debugging Session in CDB

12/5/2018 • 2 minutes to read • Edit Online

You can exit CDB by entering the **q (Quit)** command. This command also closes the application that you are debugging.

The **qd (Quit and Detach)** command detaches CDB from the target application, exits the debugger, and leaves the target application running. If you used the **-pd** command-line option when you started the debugger, detaching occurs if the session is ended for any reason. (This technique makes **-pd** especially useful when you are debugging a sensitive process, such as the Client Server Run-Time Subsystem (CSRSS), that you do not want to end.)

If the debugger is not responding, you can exit by pressing **CTRL+B** and then ENTER. This method is a secondary exit mechanism. It abruptly ends the debugger and is similar to ending a process through Task Manager or by closing the window.

To end a user-mode debugging session, return the debugger to dormant mode, and close the target application, you can use the following method:

- Enter the **.kill (Kill Process)** command.

To end a user-mode debugging session, return the debugger to dormant mode, and set the target application running again, you can use the following methods:

- Enter the **.detach (Detach from Process)** command. If you are debugging multiple targets, this command detaches from the current target and continues the debugging session with the remaining targets.
- Enter the **qd (Quit and Detach)** command.
- Enter the **q (Quit)** command, if you started the debugger with the **-pd** option.

To end a user-mode debugging session, return the debugger to dormant mode, but leave the target application in the debugging state, you can use the following method:

- Enter the **.abandon (Abandon Process)** command.

For more information about reattaching to the target, see Reattaching to the Target Application.

Setting Symbol and Executable Image Paths in CDB

12/5/2018 • 2 minutes to read • Edit Online

Symbol Path

The symbol path specifies the directories where the symbol files are located. For more information about symbols and symbol files, see Symbols.

Note If you are connected to the Internet or a corporate network, the most efficient way to access symbols is to use a symbol server. You can use a symbol server by using the srv* or symsrv* string within your symbol path. For more information about symbol servers, see Symbol Stores and Symbol Servers.

To control the symbol path in CDB, do one of the following:

- Enter the **.sympath (Set Symbol Path)** command. If you are using a symbol server, the **.symfix (Set Symbol Store Path)** command is similar to .sympath but saves you typing.
- When you start the debugger, use the **-y** command-line option. See **CDB Command-Line Options**.
- Before you start the debugger, use the _NT_SYMBOL_PATH and _NT_ALT_SYMBOL_PATH environment variables to set the path. The symbol path is created by appending _NT_SYMBOL_PATH after _NT_ALT_SYMBOL_PATH. (Typically, the path is set through the _NT_SYMBOL_PATH. However, you might want to use _NT_ALT_SYMBOL_PATH to override these settings in special cases, such as when you have private versions of shared symbol files.)

  Note If you use the **-sins** command-line option, the debugger ignores the symbol path environment variable.

Executable Image Path

An executable file is a binary file that the processor can run. These files typically have the .exe, .dll, or .sys file name extension. Executable files are also known as modules, especially when executable files are described as units of a larger application. Before the Windows operating system runs an executable file, it loads it into memory. The copy of the executable file in memory is called the executable image or the image.

Note These terms are sometimes used imprecisely. For example, some documents might use "image" for the actual file on the disk. Also, the Windows kernel and HAL have special module names. For example, the **nt** module corresponds to the Ntoskrnl.exe file.

The executable image path specifies the directories that the binary executable files are located in.

In most situations, the debugger knows the location of the executable files, so you do not have to set the path for this file.

However, there are situations when this path is required. For example, kernel-mode small memory dump files do not contain all of the executable files that exist in memory at the time of a stop error (that is, a crash). Similarly, user-mode minidump files do not contain the application binaries. If you set the path of the executable files, the debugger can find these binary files.

The debugger's executable image path is a string that consists of multiple directory paths, separated by semicolons. Relative paths are supported. However, unless you always start the debugger from the same directory, you should add a drive letter or a network share before each path. Network shares are also supported. The debugger searches the executable image path recursively. That is, the debugger searches the subdirectories of each

directory that is listed in this path.

To control the executable image path in CDB, do one of the following:

- Enter the **.exepath (Set Executable Path)** command.
- When you start the debugger, use the **-i** command-line option. See **CDB Command-Line Options**.
- Before you start the debugger, use the _NT_EXECUTABLE_IMAGE_PATH environment variable to set the path.

  Note If you use the **-sins** command-line option, the debugger ignores the executable image path environment variable.

Setting Breakpoints in CDB

12/5/2018 • 2 minutes to read • Edit Online

You can set, view, and manipulate breakpoints in CDB by entering commands. For a list of commands, see Methods of Controlling Breakpoints.

Viewing the Call Stack in CDB

12/5/2018 • 2 minutes to read • Edit Online

The call stack is the chain of function calls that have led to the current location of the program counter. The top function on the call stack is the current function, the next function is the function that called the current function, and so on. The call stack that is displayed is based on the current program counter, unless you change the register context. For more information about how to change the register context, see Changing Contexts.

In CDB, you can view the call stack by entering one of the **k (Display Stack Backtrace)** commands.

Viewing and Editing Memory in CDB

12/5/2018 • 2 minutes to read • Edit Online

Viewing and Editing Memory

In CDB, you can view and edit memory by entering one of the **Display Memory** commands, and you can edit memory by entering one of the **Enter Values** commands. For a detailed discussion of these commands, see Accessing Memory by Virtual Address and Accessing Memory by Physical Address.

Viewing and Editing Variables

In CDB, you can view and edit global variables by entering commands. The debugger interprets the name of a global variable as a virtual address. Therefore, all of the commands that are described in Accessing Memory by Virtual Address can be used to read or write global variables. For additional information about viewing and editing global variables, see Accessing Global Variables.

In CDB you can view and edit local variables by entering commands. The debugger interprets the name of a local variable as an address. Therefore, all of the commands that are described in Accessing Memory by Virtual Address can be used to read or write local variables. However, if it is necessary to indicate to a command that a symbol is local, precede the symbol with a dollar sign ($) and an exclamation point (!), as in `$!var`. For additional information about viewing and editing local variables, see Accessing Local Variables.

Viewing and Editing Registers in CDB

12/5/2018 • 2 minutes to read • Edit Online

Registers are small volatile memory units that are located on the CPU. Many registers are dedicated to specific uses, and other registers are available for user-mode applications to use. The x86-based and x64-based processors have different collections of registers available. For more information about the registers on each processor, see Processor Architecture.

In CDB, you can view registers by entering the **r (Registers)** command in the Debugger Command window. You can customize the display by using several options or by using the **rm (Register Mask)** command.

Registers are also automatically displayed every time that the target stops. If you are stepping through your code with the **p (Step)** or **t (Trace)** commands, you see a register display at every step. To stop this display, use the **r** option when you use these commands.

On an x86-based processor, the **r** option also controls several one-bit registers known as flags. To change these flags, you use a slightly different syntax than when changing regular registers. For more information about these flags and an explanation of this syntax, see x86 Flags.

Configuring Exceptions and Events in CDB

12/5/2018 • 2 minutes to read • Edit Online

You can configure CDB to react to specified exceptions and events in a specific way. For each exception, you can set the break status and the handling status. For each event, you can set the break status.

You can configure the break status or handling status by doing one of the following:

- Use the **SXE**, **SXD**, **SXN**, or **SXI** command.
- Use the **-x**, **-xe**, **-xd**, **-xn**, or **-xi** option on the **CDB command line**.
- Use the **sxe** or **sxd** keyword in the Tools.ini file.

For a detailed discussion of exceptions and events, see Controlling Exceptions and Events.

Keeping a Log File in CDB

12/5/2018 • 2 minutes to read • Edit Online

CDB can write a log file that records the debugging session. This log file contains all of the commands that you type and the responses from the debugger.

Opening a New Log File

To open a new log file, or to overwrite a previous log file, do one of the following:

- Before you start the debugger, set the _NT_DEBUG_LOG_FILE_OPEN environment variable.
- When you start the debugger, use the **-logo** command-line option.
- Enter the **.logopen (Open Log File)** command. If you use the **/t** option, the date and time are appended to your specified file name. If you use the **/u** option, the log file is written in Unicode instead of in ASCII.

Appending to an Existing Log File

To append command window text to a log file, do one of the following:

- Before you start the debugger, set the _NT_DEBUG_LOG_FILE_APPEND environment variable.
- When you start the debugger, use the **-loga** command-line option.
- Enter the **.logappend (Append Log File)** command. If you are appending to a Unicode log file, you must use the **/u** option.

Closing a Log File

To close an open log file, do the following:

- Enter the **.logclose (Close Log File)** command.

Checking Log File Status

To determine the log file status, do the following:

- Enter the **.logfile (Display Log File Status)** command.

Local Kernel-Mode Debugging

12/5/2018 • 2 minutes to read • Edit Online

Debugging Tools for Windows supports *local kernel debugging*. This is kernel-mode debugging on a single computer. In other words, the debugger runs on the same computer that is being debugged.

Setting Up Local Kernel-Mode Debugging

For information on setting up local kernel-mode debugging, see Setting Up Local Kernel-Mode Debugging of a Single Computer Manually.

Starting the Debugging Session

Using WinDbg

Open WinDbg as Administrator. On the **File** menu, choose **Kernel Debug**. In the Kernel Debugging dialog box, open the **Local** tab. Click **OK**.

You can also start a session with WinDbg by opening a Command Prompt window as Administrator and entering the following command:

windbg -kl

Using KD

Open a Command Prompt window as Administrator, and enter the following command:

kd -kl

Commands That Are Not Available

Not all commands are available in a local kernel debugging session. Typically, you cannot use any command that causes the target computer to stop, even momentarily, because you cannot resume operation.

In particular, you cannot use the following commands:

- Execution commands, such as **g (Go)**, **p (Step)**, **t (Trace)**, **wt (Trace and Watch Data)**, **tb (Trace to Next Branch)**, **gh (Go with Exception Handled)**, and **gn (Go with Exception Not Handled)**
- Shutdown and dump file commands, such as **.crash**, **.dump**, and **.reboot**
- Breakpoint commands, such as **bp**, **bu**, **ba**, **bc**, **bd**, **be**, and **bl**
- Register display commands, such as **r** and variations
- Stack trace commands, such as **k** and variations

If you are performing local kernel debugging with WinDbg, all of the equivalent menu commands and buttons are also unavailable.

Commands That Are Available

All memory input and output commands are available. You can freely read from user memory and kernel memory. You can also write to memory. Make sure that you do not write to the wrong part of kernel memory, because it can corrupt data structures and frequently causes the computer to stop responding (that is, *crash*).

Difficulties in Performing Local Kernel Debugging

Local kernel debugging is a very delicate operation. Be careful that you do not corrupt or crash the system.

One of the most difficult aspects of local kernel debugging is that the machine state is constantly changing. Memory is paged in and out, the active process constantly changes, and virtual address contexts do not remain constant. However, under these conditions, you can effectively analyze things that change slowly, such as certain device states.

Kernel-mode drivers and the Windows operating system frequently send messages to the kernel debugger by using **DbgPrint** and related functions. These messages are not automatically displayed during local kernel debugging. You can display them by using the **!dbgprint** extension.

LiveKD

The LiveKD tool simulates local kernel debugging. This tool creates a "snapshot" dump file of the kernel memory, without actually stopping the kernel while this snapshot is made. (Therefore, the snapshot might not actually show a single instant state of the computer.)

LiveKD is not part of the Debugging Tools for Windows package. You can download LiveKd from the Windows Sysinternals site.

Controlling the Target

12/5/2018 • 6 minutes to read • Edit Online

While you are debugging a target application in user mode or a target computer in kernel mode, the target can be *running* or *stopped*.

When the debugger connects to a kernel-mode target, the debugger leaves the target running, unless you use the **-b** command-line option, the target system has stopped responding (that is, *crashed*), or the target system is still stopped because of an earlier kernel debugging action.

When the debugger starts or connects to a user-mode target, the debugger immediately stops the target, unless you use the **-g** command-line option. For more information, see Initial Breakpoint.

When the Target is Running

When the target is running, most debugger actions are unavailable.

If you want to stop a running target, you can issue a **Break** command. This command causes the debugger to *break into the target*. That is, the debugger stops the target and all control is given to the debugger. The application might not break immediately. For example, if all threads are currently executing system code, or are in a wait operation, the application breaks only after control has returned to the application's code.

If a running target encounters an exception, if certain events occur, if a breakpoint is hit, or if the application closes normally, the target *breaks into the debugger*. This action stops the target and gives all control to the debugger. A message appears in the Debugger Command window and describes the error, event, or breakpoint.

When the Target is Stopped

To start or control the target's execution, you can do the following:

- To cause the application to begin running, issue the **Go** command.
- To step through the application one instruction at a time, use the **Step Into** or **Step Over** commands. If a function call occurs, **Step Into** enters the function and continues stepping through each instruction. **Step Over** treats the function call as a single step. When the debugger is in Assembly Mode, stepping occurs one machine instruction at a time. When the debugger is in Source Mode, stepping occurs one source line at a time.
- To finish the current function and stop when the return occurs, use the **Step Out** or **Trace and Watch** commands. The **Step Out** command continues until the current function ends. **Trace and Watch** continues until the current function ends and also displays a summary of the function's calls. However, you must issue the **Trace and Watch** command on the first instruction of the function in question.
- If an exception occurs, you can use the **Go with Exception Handled** and **Go with Exception Not Handled** commands to resume execution and control the status of the exception. (For more information about exceptions, see Controlling Exceptions and Events.)
- (WinDbg only) If you select a line in the Disassembly window or a Source window and then use the **Run to Cursor** command, the program runs until it encounters the selected line.
- (User Mode only) To close the target application and restart it from the beginning, use the **Restart** command. You can use this command only with a process that the debugger created. After the process is restarted, it immediately breaks into the debugger.
- (WinDbg only) To close the target application and clear the debugger, use the **Stop Debugging**

command. This command enables you to start debugging a different target.

Command Forms

Most commands for starting or controlling the target's execution exist as text commands, menu commands, toolbar buttons, and shortcut keys. As basic text commands, you can use these commands in CDB, KD, or WinDbg. (The text form of the commands frequently supports additional options, such as changing the location of the program counter or executing a fixed number of instructions.) You can use the menu commands, toolbar buttons, and shortcut keys in WinDbg.

You can use the commands in the following forms.

COMMAND	WINDBG BUTTON	WINDBG COMMAND	WINDBG SHORTCUT KEYS	EFFECT
		Debug | Run to Cursor	F7 CTRL + F10	(WinDbg only) Executes until it reaches the line that the cursor marks.
		Debug | Stop Debugging	SHIFT + F5	Stops all debugging and closes the target.
(CDB/KD only) **CTRL+C**		Debug | Break	CTRL + BREAK	Execution stops, and the debugger breaks into the target.
.restart (Restart Target Application)		Debug | Restart	CTRL + SHIFT + F5	(User mode only) Restarts the target application.
g (Go)		Debug | Go	F5	Target executes freely.
gc (Go from Conditional Breakpoint)				Resumes execution after a conditional breakpoint.
gh (Go with Exception Handled)		Debug | Go Handled Exception		Same as **g (Go)**, except that the current exception is treated as handled.

COMMAND	WINDBG BUTTON	WINDBG COMMAND	WINDBG SHORTCUT KEYS	EFFECT
gn (Go with Exception Not Handled)		Debug | Go Unhandled Exception		Same as **g (Go)**, except that the current exception is treated as unhandled.
gu (Go Up)		Debug | Step Out	SHIFT + F11	Target executes until the current function is complete.
p (Step)		Debug | Step Over	F10	Target executes one instruction. If this instruction is a function call, that function is executed as a single step.
pa (Step to Address)				Target executes until it reaches the specified address. All steps in this function are displayed (but steps in called functions are not).
pc (Step to Next Call)				Target executes until the next **call** instruction. If the current instruction is a **call** instruction, this call is executed completely and execution continues until the next **call**.
pct (Step to Next Call or Return)				Target executes until it reaches a **call** instruction or a **return** instruction.

COMMAND	WINDBG BUTTON	WINDBG COMMAND	WINDBG SHORTCUT KEYS	EFFECT
ph (Step to Next Branching Instruction)				Target executes until it reaches any kind of branching instruction, including conditional or unconditional branches, calls, returns, and system calls.
pt (Step to Next Return)				Target executes until it reaches a **return** instruction.
t (Trace)		Debug | Step Into	F11 F8	Target executes one instruction. If this instruction is a function call, debugger traces into that call.
ta (Trace to Address)				Target executes until it reaches the specified address. All steps in this function and called functions are displayed.
tb (Trace to Next Branch)				(All modes, except kernel mode, only on x86-based systems) Target executes until it reaches the next branch instruction.
tc (Trace to Next Call)				Target executes until the next **call** instruction. If the current instruction is a **call** instruction, the instruction is traced into until a new **call** is reached.

COMMAND	WINDBG BUTTON	WINDBG COMMAND	WINDBG SHORTCUT KEYS	EFFECT
tct (Trace to Next Call or Return)				Target executes until it reaches a **call** instruction or **return** instruction. If the current instruction is a **call** instruction or **return** instruction, the instruction is traced into until a new **call** or **return** is reached.
th (Trace to Next Branching Instruction)				Target executes until it reaches any kind of branching instruction, including conditional or unconditional branches, calls, returns, and system calls. If the current instruction is a branching instruction, the instruction is traced into until a new branching instruction is reached.
tt (Trace to Next Return)				Target executes until it reaches a **return** instruction. If the current instruction is a **return** instruction, the instruction is traced into until a new **return** is reached.

COMMAND	WINDBG BUTTON	WINDBG COMMAND	WINDBG SHORTCUT KEYS	EFFECT
wt (Trace and Watch Data)				Target executes until the completion of the whole specified function. Statistics are then displayed.

For more information about how to restart the target computer, see Crashing and Rebooting the Target Computer.

Command-Line Options

If you do not want the application to stop immediately when it starts or loads, use CDB or WinDbg together with the **-g** command-line option. For more information about this situation, see Initial Breakpoint.

CDB and WinDbg also support the **-G** command-line option. This option causes the debugging session to end if the application completes properly.

The following command tries to run the application from start to finish, and the debugger prompt appears only if an error occurs.

```
cdb -g -G ApplicationName
```

You can use the **-pt** command-line option to set the break time-out. There are certain problems that can make the target unable to communicate with the debugger. If a break command is issued and the debugger cannot break into the target after this time, the debugger displays a "Break-in timed out" message.

At this point, the debugger stops trying to break into the target. Instead, the debugger pauses the target and enables you to examine (but not control) the target application.

The default time-out is 30 seconds.

Enabling Postmortem Debugging

6/25/2019 • 12 minutes to read • Edit Online

User Mode Exception Handling

Exceptions and Breakpoints

The most common application errors are called exceptions. These include access violations, division-by-zero errors, numerical overflows, CLR exceptions, and many other kinds of errors. Applications can also cause breakpoint interrupts. These occur when Windows is unable to run the application (for example, when a necessary module cannot be loaded) or when a breakpoint is encountered. Breakpoints can be inserted into the code by a debugger, or invoked through a function such as **DebugBreak**.

Exception Handlers Precedence

Based on configuration values and which debuggers are active, Windows handles user-mode errors in a variety of ways. The following sequence shows the precedence used for user mode error handling:

1. If a user-mode debugger is currently attached to the faulting process, all errors will cause the target to break into this debugger.

   As long as the user-mode debugger is attached, no other error-handling methods will be used -- even if the **gn (Go With Exception Not Handled)** command is used.

2. If no user-mode debugger is attached and the executing code has its own exception handling routines (for example, **try - except**), this exception handling routine will attempt to deal with the error.

3. If no user-mode debugger is attached, and Windows has an open kernel-debugging connection, and the error is a breakpoint interrupt, Windows will attempt to contact the kernel debugger.

   Kernel debugging connections must be opened during Windows' boot process. If you wish to prevent a user-mode interrupt from breaking into the kernel debugger, you can use the KDbgCtrl utility with the **-du** parameter. For details on how to configure kernel-debugging connections and how to use KDbgCtrl, see Getting Set Up for Debugging.

   In the kernel debugger, you can use **gh (Go With Exception Handled)** to disregard the error and continue running the target. You can use **gn (Go With Exception Not Handled)** to bypass the kernel debugger and go on to step 4.

4. If the conditions in steps 1, 2, and 3 do not apply, Windows will activate a debugging tool configured in the AeDebug registry values. Any program can be selected in advance as the tool to use in this situation. The chosen program is referred to as the *postmortem debugger*.

5. If the conditions in steps 1, 2, and 3 do not apply, and there is no postmortem debugger registered, Windows Error Reporting (WER) displays a message and provides solutions if any are available. WER also writes a memory dump file if the appropriate values are set in the Registry. For more information, see Using WER and Collecting User-Mode Dumps.

DebugBreak Function

If a postmortem debugger has been installed, you can deliberately break into the debugger from a user-mode application by calling the **DebugBreak** function.

Specifying a Postmortem Debugger

This section describes how to configure tools such as WinDbg as the postmortem debugger. Once configured, the postmortem debugger will be automatically started whenever an application crashes.

Post Mortem Debugger Registry Keys

Windows Error Reporting (WER) creates the postmortem debugger process using the values set in the AeDebug registry key.

HKLM\Software\Microsoft\Windows NT\CurrentVersion\AeDebug

There are two primary registry values of interest, *Debugger* and *Auto*. The *Debugger* registry value specifies the command line for the postmortem debugger. The *Auto* registry value specifies if the postmortem debugger is automatically started, or if a confirmation message box is presented first.

Debugger (REG_SZ)

This REG_SZ value specifies the debugger that will handle postmortem debugging.

The full path to the debugger must be listed unless the debugger is located in a directory that is in the default path.

The command line is generated from the Debugger string via a printf style call that includes 3 parameters. Although the order is fixed, there is no requirement to use any or all of the available parameters.

DWORD (%ld) - Process ID of the target process.

DWORD (%ld) - Event Handle duplicated into the postmortem debugger process. If the postmortem debugger signals the event, WER will continue the target process without waiting for the postmortem debugger to terminate. The event should only be signaled if the issue has been resolved. If the postmortem debugger terminates without signaling the event, WER continues the collection of information about the target processes.

void* (%p) - Address of a JIT_DEBUG_INFO structure allocated in the target process's address space. The structure contains additional exception information and context.

Auto (REG_SZ) This REG_SZ value is always either **0** or **1**.

If **Auto** is set to **0**, a confirmation message box is displayed prior to postmortem debugging process being started.

If **Auto** is set to **1**, the postmortem debugger is immediately created.

When you manually edit the registry, do so very carefully, because improper changes to the registry may not allow Windows to boot.

Example Command Line Usage

Many postmortem debuggers use a command line that includes -p and -e switches to indicate the parameters are a PID and Event (respectively). For example, installing WinDbg via `windbg.exe -I` creates the following values:

```
Debugger = "<Path>\WinDbg -p %ld -e %ld -g"
Auto = 1
```

There is flexibility in how the WER %ld %ld %p parameters can be used. For example. there is no requirement to specify any switches around or between the WER parameters. For example, installing Windows Sysinternals ProcDump using `procdump.exe -i` creates the following values with no switches between the WER %ld %ld %p parameters:

```
Debugger = "<Path>\procdump.exe" -accepteula -j "c:\Dumps" %ld %ld %p
Auto = 1
```

32 and 64 bit Debuggers

On a 64-bit platform, the Debugger (REG_SZ) and Auto (REG_SZ) registry values are defined individually for 64-bit and 32-bit applications. An additional Windows on Windows (WOW) key is used to store the 32 bit application post mortem debugging values.

HKLM\Software\Wow6432Node\Microsoft\Windows NT\CurrentVersion\AeDebug

On a 64-bit platform, use a 32-bit post-mortem debugger for 32-bit processes and a 64-bit debugger for 64-bit processes. This avoids a 64-bit debugger focusing on the WOW64 threads, instead of the 32-bit threads, in a 32-bit process.

For many postmortem debuggers, including the Debugging Tools for Windows postmortem debuggers, this involves running the installation command twice; once with the x86 version and once with the x64 version. For example, to use WinDbg as the interactive postmortem debugger, the `windbg.exe -I` command would be run twice, once for each version.

64-bit Installation:

```
C:\Program Files (x86)\Windows Kits\10\Debuggers\x64\windbg.exe –I
```

This updates the registry key with these values.

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AeDebug
Debugger = "C:\Program Files (x86)\Windows Kits\10\Debuggers\x64\windbg.exe" -p %ld -e %ld –g
```

32-bit Installation:

```
C:\Program Files (x86)\Windows Kits\10\Debuggers\x86\windbg.exe –I
```

This updates the registry key with these values.

```
HKEY_LOCAL_MACHINE\SOFTWARE\WOW6432Node\Microsoft\Windows NT\CurrentVersion\AeDebug
Debugger = "C:\Program Files (x86)\Windows Kits\10\Debuggers\x86\windbg.exe" -p %ld -e %ld –g
```

Configuring Post Mortem Debuggers

Debugging Tools for Windows

The Debugging Tools for Windows debuggers all support being set as the postmortem debugger. The install command intends for the process to be debugged interactively.

WinDbg

To set the postmortem debugger to WinDbg, run `windbg -I`. (The `I` must be capitalized.) This command will display a success or failure message after it is used. To work with both 32 and 64 bit applications, run the command for the both the 64 and 32 debuggers.

```
C:\Program Files (x86)\Windows Kits\10\Debuggers\x64\windbg.exe –I
C:\Program Files (x86)\Windows Kits\10\Debuggers\x86\windbg.exe –I
```

This is how the AeDebug registry entry will be configured when `windbg -I` is run.

```
Debugger = "<Path>\WinDbg -p %ld -e %ld -g"
Auto = 1
```

In the examples, <*Path*> is the directory where the debugger is located.

The -p and -e parameters pass the Process ID and Event, as discussed previously.

The **-g** passes the g (Go) command to WinDbg and continues execution from the current instruction.

Note There is a significant issue passing the g (Go) command. The issue with this approach, is that exceptions do not always repeat, typically, because of a transient condition that no longer exists when the code is restarted. For more information about this issue, see **.jdinfo (Use JIT_DEBUG_INFO)**.

To avoid this issue, use .jdinfo or .dump /j. This approach allows the debugger to be in the context of the code failure of interest. For more information, see Just In Time (JIT) Debugging later in this topic.

CDB

To set the postmortem debugger to CDB, run **cdb -iae** (Install AeDebug) or **cdb -iaec** *KeyString* (Install AeDebug with Command).

```
C:\Program Files (x86)\Windows Kits\10\Debuggers\x64\cdb.exe -iae
C:\Program Files (x86)\Windows Kits\10\Debuggers\x86\cdb.exe -iae
```

When the **-iaec** parameter is used, *KeyString* specifies a string to be appended to the end of command line used to launch the postmortem debugger. If *KeyString* contains spaces, it must be enclosed in quotation marks.

```
C:\Program Files (x86)\Windows Kits\10\Debuggers\x64\cdb.exe -iaec [KeyString]
C:\Program Files (x86)\Windows Kits\10\Debuggers\x86\cdb.exe -iaec [KeyString]
```

This command display nothing if it succeeds, and an error message if it fails.

NTSD

To set the postmortem debugger to NTSD, run **ntsd -iae** (Install AeDebug) or **ntsd -iaec** *KeyString* (Install AeDebug with Command).

```
C:\Program Files (x86)\Windows Kits\10\Debuggers\x64\ntsd.exe -iae
C:\Program Files (x86)\Windows Kits\10\Debuggers\x86\ntsd.exe -iae
```

When the **-iaec** parameter is used, *KeyString* specifies a string to be appended to the end of command line used to launch the postmortem debugger. If *KeyString* contains spaces, it must be enclosed in quotation marks.

```
C:\Program Files (x86)\Windows Kits\10\Debuggers\x64\ntsd.exe -iaec [KeyString]
C:\Program Files (x86)\Windows Kits\10\Debuggers\x86\ntsd.exe -iaec [KeyString]
```

This command display nothing if it succeeds, and an error to a new console window on failure.

Note Because the -p %ld -e %ld -g parameters always appear first on the command line of the postmortem debugger, you should not use the -iaec switch to specify the -server parameter because -server will not work unless it appears first on the command line. To install a postmortem debugger that includes this parameter, you must edit the registry manually.

Visual Studio JIT Debugger

If Visual Studio has been installed, vsjitdebugger.exe will be registered as the post mortem debugger. The Visual Studio JIT Debugger intends for the process to be debugged interactively.

```
Debugger = "C:\WINDOWS\system32\vsjitdebugger.exe" -p %ld -e %ld
```

If Visual Studio is updated or re-installed, this entry will be re-written, overwriting any alternate values set.

Window Sysinternals ProcDump

The Windows Sysinternals ProcDump utility can also be used for postmortem dump capture. For more information about using and downloading ProcDump, see ProcDump on TechNet.

Like the **.dump** WinDbg command, ProcDump is able to be capture a dump of the crash non-interactively. The capture may occur in any Windows system session.

ProcDump exits when the dump file capture completes, WER then reports the failure and the faulting process is terminated.

Use `procdump -i` to install procdump and -u to uninstall ProcDump for both the 32 and 64 bit post mortem debugging.

```
<Path>\procdump.exe -i
```

The install and uninstall commands output the registry values modified on success, and the errors on failure.

The ProcDump command line options in the registry are set to:

```
Debugger = <Path>\ProcDump.exe -accepteula -j "<DumpFolder>" %ld %ld %p
```

ProcDump uses all 3 parameters - PID, Event and JIT_DEBUG_INFO. For more information on the JIT_DEBUG_INFO parameter, see Just In Time (JIT) Debugging below.

The size of dump captured defaults to Mini (process/threads/handles/modules/address space) without a size option set, MiniPlus (Mini plus MEM_PRIVATE pages) with -mp set, or Full (all memory - equivalent to ".dump /mA") with -ma set.

For systems with sufficient drive space, a Full (-ma) capture is recommended.

Use -ma with the -i option to specify an all memory capture. Optionally, provide a path for the dump files.

```
<Path>\procdump.exe -ma -i c:\Dumps
```

For systems with limited drive space, a MiniPlus (-mp) capture is recommended.

```
<Path>\procdump.exe -mp -i c:\Dumps
```

The folder to save the dump file to is optional. The default is the current folder. The folder should secured with an ACL that is equal or better than what is used for C:\Windows\Temp. For more information on managing security related to folders, see Security During Postmortem Debugging.

To uninstall ProcDump as the postmortem debugger, and restore the previous settings, use the -u (Uninstall) option.

```
<Path>\procdump.exe -u
```

The install and uninstall commands set both the 64-bit and 32-bit values on 64-bit platforms.

ProcDump is a "packed" executable containing both the 32-bit and 64-bit version of application - as such, the same executable is used for both 32-bit and 64-bit. When ProcDump runs, it automatically switches the version, if the version running doesn't match the target process.

Just In Time (JIT) Debugging

Setting Context to the Faulting Application

As discussed previously, it is very desirable to set the context to the exception that caused the crash using the JIT_DEBUG_INFO parameter. For more information about this, see **.jdinfo (Use JIT_DEBUG_INFO)**.

Debugging Tools for Windows

This example shows how to edit the registry to run an initial command (-c) that uses the .jdinfo <address> command to display the additional exception information, and change the context to the location of the exception (similar to how .ecxr is used set the context to the exception record).

```
Debugger = "<Path>\windbg.exe -p %ld -e %ld -c ".jdinfo 0x%p"
Auto = 1
```

The %p parameter is the address of a JIT_DEBUG_INFO structure in the target process's address space. The %p parameter is pre-appended with 0x so that it is interpreted as a hex value. For more information, see **.jdinfo (Use JIT_DEBUG_INFO)**.

To debug a mix of 32 and 64 bit apps, configure both the 32 and 64 bit registry keys (described above), setting the proper path to the location of the 64-bit and 32-bit WinDbg.exe.

Creating a dump file using .dump

To capture a dump file whenever a failure occurs that includes the JIT_DEBUG_INFO data, use .dump /j <address>.

```
<Path>\windbg.exe -p %ld -e %ld -c ".dump /j %p /u <DumpPath>\AeDebug.dmp; qd"
```

Use the /u option to generate a unique filename to allow multiple dump files to be automatically created. For more information about the options see, **.dump (Create Dump File)**.

The created dump will have the JITDEBUG_INFO data stored as the default exception context. Instead of using .jdinfo to view the exception information and set the context, use .exr -1 to display the exception record and .ecxr to set the context. For more information see **.exr (Display Exception Record)** and **.ecxr (Display Exception Context Record)**.

Windows Error Reporting - q / qd

The way the debug session ends determines if Windows Error Reporting reports the failure.

If the debug session is detached using qd prior to the closing of the debugger, WER will report the failure.

If the debug session is quit using q (or if the debugger is closed without detaching), WER will not report the failure.

Append *;q* or *;qd* to the end of the command string to invoke the desired behavior.

For example, to allow WER to report the failure after CDB captures a dump, configure this command string.

```
<Path>\cdb.exe -p %ld -e %ld -c ".dump /j 0x%p /u c:\Dumps\AeDebug.dmp; qd"
```

This example would allow WER to report the failure after WinDbg captures a dump.

```
<Path>\windbg.exe -p %ld -e %ld -c ".dump /j %p /u <DumpPath>\AeDebug.dmp; qd""
```

Security Vulnerabilities

If you are considering enabling postmortem debugging on a computer that you share with other people, see Security During Postmortem Debugging.

Using the Debugger Command Window

12/5/2018 • 2 minutes to read • Edit Online

This section includes the following topics:

Using Debugger Commands

Evaluating Expressions

Using Shell Commands

Using Aliases

Using Script Files

Using Debugger Command Programs

Using Debugger Commands

12/5/2018 • 7 minutes to read • Edit Online

For KD or CDB, "Debugger Command window" refers to the whole window. You enter commands at the prompt at the bottom of the window. If the commands have any output, the window displays the output and then displays the prompt again.

For Visual Studio, "Debugger Command window" refers to a window that is labeled "Debugger Immediate Window" in the title bar. This window has two panes:

- In the small, bottom pane, you enter commands.
- In the large, upper pane, you view command output.

For WinDbg, "Debugger Command window" refers to the window that is labeled "Command" in the title bar. This window contains two panes:

- In the small, bottom pane, you enter commands.
- In the large, upper pane, you view command output.

This window is always open at the beginning of a debugging session. You can reopen or switch to this window by clicking **Command** on the **View** menu, pressing ALT+1, or clicking the **Command (Alt+1)** button ( ) on the toolbar.

You can use the UP ARROW and DOWN ARROW keys to scroll through the command history. When a previous command appears, you can edit it and then press ENTER to execute the previous command (or the edited version of the previous command). The cursor does not have to be at the end of the line for this procedure to work correctly.

Debugger Command Window Prompt

When you are performing user-mode debugging, the prompt in the Debugger Command window looks like the following example.

```
2:005>
```

In the preceding example, 2 is the current process number, and 005 is the current thread number.

If you attach the debugger to more than one computer, the system number is included before the process and thread number, as in the following example.

```
3:2:005>
```

In this example, 3 is the current system number, 2 is the current process number, and 005 is the current thread number.

When you are performing kernel-mode debugging on a target computer that has only one processor, the prompt looks like the following example.

```
kd>
```

However, if the target computer has multiple processors, the number of the current processor appears before the prompt, as in the following example.

```
0: kd>
```

If the debugger is busy processing a previously issued command, new commands will temporarily not be processed, although they can be added to the command buffer. In addition, you can still use control keys in KD and CDB, and you can still use menu commands and shortcut keys in WinDbg. When KD or CDB is in this busy state, no prompt is displayed. When WinDbg is in this busy state, the following indicator will appear in place of the prompt:

```
*BUSY*
```

You can use the **.pcmd (Set Prompt Command)** command to add text to this prompt.

Kinds of Commands

WinDbg, KD, and CDB support a variety of commands. Some commands are shared between the debuggers, and some are available only on one or two of the debuggers.

Some commands are available only in live debugging, and other commands are available only when you debug a dump file.

Some commands are available only during user-mode debugging, and other commands are available only during kernel-mode debugging.

Some commands are available only when the target is running on certain processors. For more information about all of the commands and their restrictions, see Debugger Commands.

Editing, Repeating, and Canceling Commands

You can use standard editing keys when you enter a command:

- Use the UP ARROW and DOWN ARROW keys to find previous commands.
- Edit the current command with the BACKSPACE, DELETE, INSERT, and LEFT ARROW and RIGHT ARROW keys.
- Press the ESC key to clear the current line.

You can press the TAB key to automatically complete your text entry. In any of the debuggers, press the TAB key after you enter at least one character to automatically complete a command. Press the TAB key repeatedly to cycle through text completion options, and hold down the SHIFT key and press TAB to cycle backward. You can also use wildcard characters in the text and press TAB to expand to the full set of text completion options. For example, if you type **fo*!ba** and then press TAB, the debugger expands to the set of all symbols that start with "ba", in all modules with module names that start with "fo". As another example, you can complete all extension commands that have "prcb" in them by typing **!*prcb** and then pressing TAB.

When you use the TAB key to perform text completion, if your text fragment begins with a period (.), the text is matched to a dot command. If your text fragment begins with an exclamation point (!), the text is matched to an extension command. Otherwise, the text is matched with a symbol. When you usee the TAB key to enter symbols, pressing the TAB key completes code and type symbols and module names. If no module name is apparent, local symbols and module names are completed. If a module or module pattern is given, symbol completion completes code and type symbols from all matches.

You can right-click in the Debugger Command window to automatically paste the contents of the clipboard into the command that you are typing.

The maximum command length is 4096 characters. However, if you are controlling the user-mode debugger from the kernel debugger, the maximum line length is 512 characters.

In CDB and KD, press the ENTER key by itself to repeat the previous command. In WinDbg, you can enable or disable this behavior. For more information about this behavior, see **ENTER (Repeat Last Command)**.

If the last command that you issued presents a long display and you want to cut it off, use the **CTRL+C** key in

CDB or KD. In WinDbg, use Debug | Break or press CTRL+BREAK.

In kernel-mode debugging, you can cancel commands from the keyboard of the target computer by pressing **CTRL+C**.

You can use the **.cls (Clear Screen)** command to clear all of the text from the Debugger Command window. This command clears the whole command history. In WinDbg, you can clear the command history by using the Edit | Clear Command Output command or by clicking **Clear command output** on the shortcut menu of the Debugger Command window.

Expression Syntax

Many commands and extension commands accept *expressions* as their arguments. The debugger evaluates these expressions before executing the command. For more information about expressions, see Evaluating Expressions.

Aliases

Aliases are text macros that you can use to avoid having to retype complex phrases. There are two kinds of aliases. For more information about aliases, see Using Aliases.

Self-Repeating Commands

You can use the following commands to repeat an action or conditionally execute other commands:

- The **j (Execute If-Else)** conditional command
- The **z (Execute While)** conditional command
- The **~e (Thread-Specific Command)** command qualifier
- The **!list** extension command

For more information about each command, see the individual command topics.

Controlling Scrolling

You can use the scrollbar to view your previous commands and their output.

When you are using CDB or KD, any keyboard entry automatically scrolls down the Debugger Command window back to the bottom.

In WinDbg, the display automatically scrolls down to the bottom whenever a command produces output or you press the ENTER key. If you want to disable this automatic scrolling, click the Options on the **View** menu and then clear the **Automatically scroll** check box.

WinDbg Text Features

In WinDbg, you can use several additional features to change how text is displayed in the Debugger Command window. You can access some of these features in the WinDbg window, some in the shortcut menu in the Debugger Command window, and some by clicking on the appropriate menu icon.

- The **Word wrap** command on the shortcut menu turns on and off the word wrap status. This command affects the whole window, not only commands that you use after this state is changed. Because many commands and extensions produce formatted displays, we typically do not recommend word wrap.
- The Edit | Add to Command Output menu command adds a comment in the Debugger Command window. The **Add to command output** command on the shortcut menu has the same effect.
- You can customize the colors that are used for the text and the background of the Debugger Command window. You can specify different colors for different kinds of text. For example, you can display the automatic register output in one color, error messages in another color, and **DbgPrint** messages in a third color. For more information about this customization, see View | Options.
- You can use all of the features common to WinDbg's debugging information windows, such as

customizing the fonts and using special editing commands. For more information about these features, see Using Debugging Information Windows.

Remote Debugging

When you are performing remote debugging through the debugger, the debugging client can access a limited number of commands. To change the number of commands that the client can access, use the **-clines** command-line option or the _NT_DEBUG_HISTORY_SIZE environment variable.

Evaluating Expressions

12/5/2018 • 4 minutes to read • Edit Online

The debugger understands two different forms of expressions: *MASM expressions* and *C++ expressions*.

Microsoft Macro Assembler (MASM) expressions are used in the examples in this Help documentation, except when otherwise noted. In MASM expressions, all symbols are treated as addresses.

C++ expressions are the same as those used in actual C++ code. In these expressions, symbols are understood as the appropriate data types.

When Each Syntax is Used

You can select the default expression evaluator in one of the following ways:

- Use the _NT_EXPR_EVAL environment variable before the debugger is started.
- Use the **-ee** {**masm**|**c++**} command-line option when the debugger is started.
- Use the **.expr (Choose Expression Evaluator)** command to display or change the expression evaluator after the debugger is running.

If you do not use one of the preceding methods, the debugger uses the MASM expression evaluator.

If you want to evaluate an expression without changing the debugger state, you can use the **? (Evaluate Expression)** command.

All commands and debugging information windows interpret their arguments through the default expression evaluator, with the following exceptions:

- The **?? (Evaluate C++ Expression)** command always uses the C++ expression evaluator.
- The Watch window always uses the C++ expression evaluator.
- The Locals window always uses the C++ expression evaluator.
- Some extension commands always use the MASM expression evaluator (and other extension commands accept only numeric arguments instead of full expressions).
- If any part of an expression is enclosed in parentheses and you insert two at signs (**@@**) before the expression, the expression is evaluated by the expression evaluator that would not typically be used in this case.

The two at signs (**@@**) enable you to use two different evaluators for different parameters of a single command. It also enables you to evaluate different pieces of a long expression by different methods. You can nest the two at signs. Each appearance of the two at signs switches to the other expression evaluator.

Warning C++ expression syntax is useful for manipulating structures and variables, but it is not well-suited as a parser for the parameters of debugger commands. When you are using debugger commands for general purposes or you are using debugger extensions, you should set MASM expression syntax as the default expression evaluator. If you must have a specific parameter use C++ expression syntax, use the two at sign (**@@**) syntax.

For more information about the two different expression types, see Numerical Expression Syntax.

Numbers in Expressions

Numbers in MASM expressions are interpreted according to the current radix. The **n (Set Number Base)**

command can be used to set the default radix to 16, 10, or 8. All un-prefixed numbers will be interpreted in this base. The default radix can be overridden by specifying the **0x** prefix (hexadecimal), the **0n** prefix (decimal), the **0t** prefix (octal), or the **0y** prefix (binary).

Numbers in C++ expressions are interpreted as decimal numbers unless you specify differently. To specify a hexadecimal integer, add **0x** before the number. To specify an octal integer, add**0** (zero) before the number. (However, in the debugger's *output*, the **0n** decimal prefix is sometimes used.)

If you want to display a number in several bases at the same time, use the **.formats (Show Number Formats)** command.

Symbols in Expressions

The two types of expressions interpret symbols differently:

- In MASM expressions, each symbol is interpreted as an address. Depending on what the symbol refers to, this address is the address of a global variable, local variable, function, segment, module, or any other recognized label.
- In C++ expressions, each symbol is interpreted according to its type. Depending on what the symbol refers to, it might be interpreted as an integer, a data structure, a function pointer, or any other data type. A symbol that does not correspond to a C++ data type (such as an unmodified module name) creates a syntax error.

If a symbol might be ambiguous, precede it with the module name and an exclamation point (**!**). If the symbol name could be interpreted as a hexadecimal number, precede it with the module name and an exclamation point (**!**) or only an exclamation point. In order to specify that a symbol is meant to be local, omit the module name, and include a dollar sign and an exclamation point (**$!**) before the symbol name. For more information about interpreting symbols, see Symbol Syntax and Symbol Matching.

Operators in Expressions

Each expression type uses a different collection of operators.

For more information about the operators that you can use in MASM expressions and their precedence rules, see MASM Numbers and Operators.

For more information about the operators that you can use in C++ expressions and their precedence rules, see C++ Numbers and Operators.

Remember that MASM operations are always byte-based, and C++ operations follow C++ type rules (including the scaling of pointer arithmetic).

For some examples of the different syntaxes, see Expression Examples.

Using Shell Commands

12/5/2018 • 2 minutes to read • Edit Online

The debugger can transmit certain commands to the Microsoft Windows environment in which the debugger is running.

You can use the **.shell (Command Shell)** command in any Windows debugger. With this command, you can execute an application or a Microsoft MS-DOS command directly from the debugger. If you are performing remote debugging, these shell commands are executed on the server.

The **.noshell (Prohibit Shell Commands)** command or the **-noshell** command-line option disables all shell commands. The commands are disabled while the debugger is running, even if you begin a new debugging session. The commands remain disabled even if you issue a **.restart (Restart Kernel Connection)** command in KD.

If you are running a debugging server,you might want to disable shell commands. If the shell is available, a remote connection can use the **.shell** command to change your computer.

Network Drive Control

In WinDbg, you can use the File | Map Network Drive and File | Disconnect Network Drive commands to control the network drive mappings. These changes always occur on the computer that WinDbg is running on, never on any computer that is remotely connected to WinDbg.

Using Aliases

3/5/2019 • 9 minutes to read • Edit Online

Aliases are character strings that are automatically replaced with other character strings. You can use them in debugger commands and to avoid retyping certain common phrases.

An alias consists of an *alias name* and an *alias equivalent*. When you use an alias name as part of a debugger command, the name is automatically replaced by the alias equivalent. This replacement occurs immediately, before the command is parsed or executed.

The debugger supports three kinds of aliases:

- You can set and name *user-named aliases*.
- You can set *fixed-name aliases*, but they are named **$u0**, **$u1**, ..., **$u9**.
- The debugger sets and names *automatic aliases*.

Defining a User-Named Alias

When you define a user-named alias, you can choose the alias name and the alias equivalent:

- The alias name can be any string that does not contain white space.
- The alias equivalent can be any string. If you enter it at the keyboard, the alias equivalent cannot contain leading spaces or carriage returns. Alternatively, you can set it equal to a string in memory, the value of a numeric expression, the contents of a file, the value of an environment variable, or the output of one or more debugger commands.

Both the alias name and the alias equivalent are case sensitive.

To define or redefine a user-named alias, use the **as (Set Alias)** or **aS (Set Alias)** command.

To remove an alias, use the **ad (Delete Alias)** command.

To list all current user-named aliases, use the **al (List Aliases)** command.

Defining a Fixed-Name Alias

There are 10 fixed-name aliases. Their alias names are **$u0**, **$u1**, ..., **$u9**. Their alias equivalents can be any string that does not contain the ENTER keystroke.

Use the **r (Registers)** command to define the alias equivalents for fixed-name aliases. When you define a fixed-name alias, you must insert a period (**.**) before the letter "u". The text after the equal sign (=) is the alias equivalent. The alias equivalent can include spaces or semicolons, but leading and trailing spaces are ignored. You should not enclose the alias equivalent in quotation marks (unless you want quotation marks in the results).

Note Do not be confused by using the **r (Registers)** command for fixed-name aliases. These aliases are not registers or pseudo-registers, even though you use the **r** command to set their alias equivalents. You do not have to add an at (**@**) sign before these aliases, and you cannot use the **r** command to *display* the value of one of these aliases.

By default, if you do not define a fixed-name alias, it is an empty string.

Automatic Aliases

The debugger sets the following automatic aliases.

ALIAS NAME	ALIAS EQUIVALENT
$ntnsym	The most appropriate module for NT symbols on the computer's native architecture. This alias can equal either **ntdll** or **nt**.
$ntwsym	The most appropriate module for NT symbols during 32-bit debugging that uses WOW64. This alias could be **ntdll32** or some other 32-bit version of Ntdll.dll.
$ntsym	The most appropriate module for NT symbols that match the current machine mode. When you are debugging in native mode, this alias is the same as **$ntnsym**. When you are debugging in a non-native mode, the debugger tries to find a module that matches this mode. (For example, during 32-bit debugging that uses WOW64, this alias is the same as **$ntwsym**.)
$CurrentDumpFile	The name of the last dump file that the debugger loaded.
$CurrentDumpPath	The directory path of the last dump file that the debugger loaded.
$CurrentDumpArchiveFile	The name of the last dump archive file (CAB file) that the debugger loaded.
$CurrentDumpArchivePath	The directory path of the last dump archive file (CAB file) that the debugger loaded.

Automatic aliases are similar to automatic pseudo-registers, except that you can use automatic aliases with alias-related tokens (such as **${ }**), while you cannot use pseudo-registers with these tokens.

Using an Alias in the Debugger Command Window

After you define an alias, you can use it in any command entry. The alias name is automatically replaced with the alias equivalent. Therefore, you can use the alias as an expression or as a macro.

An alias name expands correctly even if it is enclosed in quotation marks. Because the alias equivalent can include any number of quotation marks or semicolons, the alias equivalent can represent multiple commands.

A user-named alias is recognized only if its name is separated from other characters by white space. The first character of its alias name must begin the line or follow a space, a semicolon, or a quotation mark. The last character of its alias name must end the line or be followed by a space, a semicolon, or a quotation mark.

Note Any text that you enter into the Debugger Command window that begins with "as", "aS", "ad", or "al" does not receive alias replacement. This restriction prevents the alias commands from being rendered inoperable. However, this restriction also means that commands that follow **ad** or **al** on a line do not have their aliases replaced. If you want aliases to be replaced in a line that begins with one of these strings, add a semicolon before the alias.

However, you can use the **${ }** token to expand a user-named alias even when it is next to other text. You can also use this token together with certain switches to prevent an alias from being expanded or to display certain alias-

related values. For more information about these situations, see **${ } (Alias Interpreter)**.

A fixed-name alias expands correctly from any point within a line, regardless of how it is embedded within the text of the line.

You cannot use commands that are available only in WinDbg (**.open**, **.write_cmd_hist (Write Command History)**, **.lsrcpath**, and **.lsrcfix**) and a few additional commands (**.hh**, **.cls**, **.wtitle**, **.remote**, kernel-mode **.restart**, and user-mode **.restart**) with aliases.

Using an Alias in a Script File

When you use an alias in a script file, you must take special care to make sure that the alias is expanded at the correct time. Consider the following script:

```
.foreach (value {dd 610000 L4})
{
   as /x ${/v:myAlias} value + 1
   .echo value myAlias
}

ad myAlias
```

The first time through the loop, the **as, aS (Set Alias)** command assigns a value to the myAlias. The value assigned to myAlias is 1 plus 610000 (the first output of the dd command). However, when the **.echo (Echo Comment)** command is executed, myAlias has not yet been expanded, so instead of seeing 610001, we see the text "myAlias".

```
0:001> $$>< c:\Script02.txt
00610000 myAlias
00905a4d 0x610001
00000003 0x905a4e
00000004 0x4
0000ffff 0x5
```

The problem is that myAlias is not expanded until a new block of code is entered. The next entry to the loop is a new block, so myAlias gets expanded to 610001. But it is too late: we should have seen 610001 the first time through the loop, not the second time.We can fix this problem by enclosing the **.echo (Echo Comment)** command in a new block as shown in the following script.

```
.foreach (value {dd 610000 L4})
{
   as /x ${/v:myAlias} value + 1
   .block{.echo value myAlias}
}

ad myAlias
```

With the altered script, we get the following correct output.

```
0:001> $$>< c:\Script01.txt
00610000 0x610001
00905a4d 0x905a4e
00000003 0x4
00000004 0x5
0000ffff 0x10000
```

For more information, see **.block** and **${ } (Alias Interpreter)**.

Using a .foreach Token in an Alias

When you use a **.foreach** token in the definition of an alias, you must take special care to ensure that the token is expanded. Consider the following sequence of commands.

```
r $t0 = 5
ad myAlias
.foreach /pS 2 /ps 2 (Token {?@$t0}) {as myAlias Token}
al
```

The first command sets the value of the **$t0** pseudo register to 5. The second command deletes any value that might have been previously assigned to myAlias. The third command takes the third token of the **?@$t0** command and attempts to assign the value of that token to myAlias. The fourth command lists all aliases and their values. We would expect the value of myAlias to be 5, but instead the value is the word "Token".

```
   Alias            Value
 -------          -------
 myAlias          Token
```

The problem is that the **as** command is at the beginning of the line in the body of the **.foreach** loop. When a line begins with an **as** command, aliases and tokens in that line are not expanded. If we put a semicolon or blank space before the **as** command, then any alias or token that already has a value is expanded. In this example, myAlias is not expanded because it does not already have a value. Token is expanded because it has a value of 5. Here is the same sequence of commands with the addition of a semicolon before the **as** command.

```
r $t0 = 5
ad myAlias
.foreach /pS 2 /ps 2 (Token {?@$t0}) {;as myAlias Token}
al
```

Now we get the expected output.

```
  Alias            Value
 -------          -------
 myAlias          5
```

Recursive Aliases

You can use a fixed-name alias in the definition of any alias. You can also use a user-named alias in the definition of a fixed-name alias. However, to use a user-named alias in the definition of another user-named alias, you have to add a semicolon before the **as** or **aS** command, or else the alias replacement does not occur on that line.

When you are using recursive definitions of this type, each alias is translated as soon as it is used. For example, the following example displays **3**, not **7**.

```
0:000> r $.u2=2
0:000> r $.u1=1+$u2
0:000> r $.u2=6
0:000> ? $u1
Evaluate expression: 3 = 00000003
```

Similarly, the following example displays **3**, not **7**.

```
0:000> as fred 2
0:000> r $.u1= 1 + fred
0:000> as fred 6
0:000> ? $u1
Evaluate expression: 3 = 00000003
```

The following example is also permitted and displays **9**.

```
0:000> r $.u0=2
0:000> r $.u0=7+$u0
0:000> ? $u0
Evaluate expression: 9 = 00000009
```

Examples

You can use aliases so that you do not have to type long or complex symbol names, as in the following example.

```
0:000> as Short usersrv!NameTooLongToWantToType
0:000> dw Short +8
```

The following example is similar to the preceding example but it uses a fixed-name alias.

```
0:000> r $.u0=usersrv!NameTooLongToWantToType
0:000> dw $u0+8
```

You can use aliases as macros for commands that you use frequently. The following example increments the **eax** and **ebx** registers two times.

```
0:000> as GoUp r eax=eax+1; r ebx=ebx+1
0:000> GoUp
0:000> GoUp
```

The following example uses an alias to simplify typing of commands.

```
0:000> as Cmd "dd esp 14; g"
0:000> bp MyApi Cmd
```

The following example is similar to the preceding example but it uses a fixed-name alias.

```
0:000> r $.u5="dd esp 14; g"
0:000> bp MyApi $u5
```

Both of the preceding examples are equivalent to the following command.

```
0:000> bp MyApi "dd esp 14; g"
```

Tools.ini File

In CDB (and NTSD), you can predefine fixed-name aliases in the tools.ini file. To predefine a fixed-name alias, add the **$u** fields that you want to your [NTSD] entry, as in the following example.

```
[NTSD]
$u1:_ntdll!_RtlRaiseException
$u2:"dd esp 14;g"
$u9:$u1 + 42
```

You cannot set user-named aliases in the Tools.ini file.

Fixed-Name Aliases vs. User-Named Aliases

User-name aliases are easier to use than fixed-named aliases. Their definition syntax is simpler, and you can list them by using the **al (List Aliases)** command.

Fixed-named aliases are replaced if they are used next to other text. To make a user-named alias be replaced when it is next to other text, enclose it in the **${ } (Alias Interpreter)** token.

Fixed-name alias replacement occurs before user-named alias replacement.

Using Script Files

12/5/2018 • 2 minutes to read • Edit Online

A *script file* is a text file that contains a sequence of debugger commands. There are a variety of ways for the debugger to load a script file and execute it. A script file can contain commands to be executed sequentially or can use a more complex flow of execution.

To execute a script file, you can do one of the following:

- (KD and CDB only; only when the debugger starts) Create a script file that is named Ntsd.ini and put it in the directory where you are starting the debugger from. The debugger automatically executes this file when the debugger starts. To use a different file for the startup script file, specify the path and file name by using the **-cf** command-line option or by using the **IniFile** entry in the Tools.ini file.
- (KD and CDB only; when each session starts) Create a script file and specify its path and file name by using the **-cfr** command-line option. The debugger automatically executes this script file when the debugger starts and every time that the target is restarted.
- Use the **$<**, **$><**, **$$<**, and **$$><** commands to execute a script file after the debugger is running. For more information about the syntax, see **$<, $><, $><, $$>< (Run Script File)**.

The **$><** and **$$><** commands differ from the other methods of running scripts in one important way. When you use these commands, the debugger opens the specified script file, replaces all carriage returns with semicolons, and executes the resulting text as a single command block. These commands are useful for running scripts that contain debugger command programs. For more information about these programs, see Using Debugger Command Programs.X

You cannot use commands that are available only in WinDbg (such as **.lsrcfix (Use Local Source Server)**, **.lsrcpath (Set Local Source Path)**, **.open (Open Source File)**, and **.write_cmd_hist (Write Command History)**) in script files, even if the script file is executed in WinDbg. In addition, you cannot use the **.beep (Speaker Beep)**, **.cls (Clear Screen)**, **.hh (Open HTML Help File)**, **.idle_cmd (Set Idle Command)**, **.remote (Create Remote.exe Server)**, kernel-mode **.restart (Restart Kernel Connection)**, user-mode **.restart (Restart Target Application)**, or **.wtitle (Set Window Title)** commands in a script file.

WinDbg supports the same scripts as KD and CDB, with one minor exception. You can use the **.remote_exit (Exit Debugging Client)** command only in a script file that KD or CDB uses. You cannot exit from a debugging client though a script that is executed in WinDbg.

Using Debugger Command Programs

12/5/2018 • 2 minutes to read • Edit Online

This section includes the following topics:

Elements of a Debugger Command Program

Control Flow Tokens

Debugger Command Program Execution

Debugger Command Program Examples

Elements of a Debugger Command Program

12/5/2018 • 2 minutes to read • Edit Online

A *debugger command program* is a small application that consists of debugger commands and control flow tokens, such as **.if**, **.for**, and **.while**. (For a full list of control flow tokens and their syntax, see Control Flow Tokens.)

You can use braces (**{ }**) to enclose a block of statements within a larger command block. When you enter each block, all aliases within the block are evaluated. If you later alter the value of an alias within a command block, commands after that point do not use the new alias value unless they are within a subordinate block.

You cannot create a block by using a pair of braces. You must add a control flow token before the opening brace. If you want to create a block only to evaluate aliases, you should use the **.block** token before the opening brace.

A debugger command program can use user-named aliases or fixed-name aliases as its local variables. If you want to use numeric or typed variables, you can use the **$t***n*pseudo-registers.

User-named aliases are evaluated only if they are not next to other text. If you want to evaluate an alias that is next to other text, use the **${ } (Alias Interpreter)** token. This token has optional switches that enable you to evaluate the alias in a variety of ways.

You can add comments to a debugger command program by using two dollar signs (**$$ (Comment Specifier)**). You should not insert a comment between a token and its elements (such as braces or conditions).

Note You should not use an asterisk (*** (Comment Line Specifier)**). Because comments that are specified with an asterisk do not end with a semicolon, the rest of the program is disregarded.

Typically, you should use MASM syntax within a debugger command program. When you have to use C++ elements (such as specifying a member of a structure or class), you can use the **@@c++( )** token to switch to C++ syntax for that clause.

The **$scmp**, **$sicmp**, and **$spat** string operators in MASM syntax are particularly useful. For more information about these operators, see MASM Numbers and Operators.

Control Flow Tokens

6/25/2019 • 2 minutes to read • Edit Online

You can use *control flow tokens* to create conditional execution and execution loops within debugger command programs.

Control flow tokens behave like their counterparts in C and C++, with the following general exceptions:

- You must enclose each block of commands that is executed conditionally or repeatedly in braces, even if there is only one such command. For example, you cannot omit the braces in the following command.

  ```
  0:000> .if (ebx>0) { r ebx }
  ```

- Each condition must be a expression. Commands are not permitted. For example, the following example produces a syntax error.

  ```
  0:000> .while (r ebx) { .... }
  ```

- The final command before a closing brace does not have to be followed by a semicolon.

The following control flow tokens are supported within a debugger command program. For more information about the syntax of each token, see the individual reference topics.

- The **.if** token behaves like the **if** keyword in C.
- The **.else** token behaves like the **else** keyword in C.
- The **.elsif** token behaves like the **else if** keyword combination in C.
- The **.foreach** token parses the output of debugger commands, a string, or a text file. This token then takes each item that it finds and uses them as the input to a specified list of debugger commands.
- The **.for** token behaves like the **for** keyword in C, except that you must separate multiple increment commands by semicolons, not by commas.
- The **.while** token behaves like the **while** keyword in C.
- The **.do** token behaves like the **do** keyword in C, except that you cannot use the word "while" before the condition.
- The **.break** token behaves like the **break** keyword in C. You can use this token within any **.for**, **.while**, or **.do** loop.
- The **.continue** token behaves like the **continue** keyword in C. You can use this token within any **.for**, **.while**, or **.do** loop.
- The **.catch** token prevents a program from ending if an error occurs. The **.catch** token is followed by braces that enclose one or more commands. If one of these commands generates an error, the error message is displayed, all remaining commands within the braces are ignored, and execution resumes with the first command after the closing brace.
- The **.leave** token is used to exit from a **.catch** block.
- The **.printf** token behaves like the **printf** statement in C.

- The **.block** token performs no action. You should use this token only to introduce a block, because you cannot create a block by only using a pair of braces. You must add a control flow token before the opening brace.

The **!for_each_module**, **!for_each_frame**, and **!for_each_local** extensions are also useful with a debugger command program.

Executing a Debugger Command Program

12/5/2018 • 2 minutes to read • Edit Online

You can execute a debugger command program in one of the following ways:

- Enter all of the statements in the Debugger Command window as a single string, with individual statements and commands separated by semicolons.
- Add all of the statements in a script file on a single line, with individual statements and commands separated by semicolons. Then, run this script file by using one of the methods described in Using Script Files.
- Add all of the statements in a script file, with each statement on a separate line. (Alternatively, separate statements by any combination of carriage returns and semicolons.) Then, run this script file by using the **$> < (Run Script File)** or **$$>< (Run Script File)** command. These commands open the specified script file, replace all carriage returns with semicolons, and execute the resulting text as a single command block.

Debugger Command Program Examples

7/9/2019 • 4 minutes to read • Edit Online

The following sections describe debugger command programs.

Using the .foreach Token

The following example uses the **.foreach** token to search for WORD values of 5a4d. For each 5a4d value that is found, the debugger displays 8 DWORD values, starting at the address of where the 5a4d DWORD was found.

```
0:000> .foreach (place { s-[1]w 77000000 L?4000000 5a4d }) { dc place L8 }
```

The following example uses the **.foreach** token to search for WORD values of 5a4d. For each 5a4d value that is found, the debugger displays 8 DWORD values, starting 4 bytes prior to the address where the 5a4d DWORD was found.

```
0:000> .foreach (place { s-[1]w 77000000 L?4000000 5a4d }) { dc place -0x4 L8 }
```

The following example displays the same values.

```
0:000> .foreach (place { s-[1]w 77000000 L?4000000 5a4d }) { dc ( place -0x4 ) L8 }
```

Note If you want to operate on the variable name in the *OutCommands* portion of the command, you must add a space after the variable name. For example, in the preceeding example, there is a space between the variable *place* and the subtraction operator.

The **-[1]** option together with the **s (Search Memory)** command causes its output to include only the addresses it finds, not the values that are found at those addresses.

The following command displays verbose module information for all modules that are located in the memory range from 0x77000000 through 0x7F000000.

```
0:000> .foreach (place { lm1m }) { .if ((${place} >= 0x77000000) & (${place} <= 0x7f000000)) { lmva place } }
```

The **1m** option together with the **lm (List Loaded Modules)** command causes its output to include only the addresses of the modules, not the full description of the modules.

The preceding example uses the **${ } (Alias Interpreter)** token to make sure aliases are replaced even if they are next to other text. If the command did not include this token, the opening parenthesis that is next to **place** prevents alias replacement. Note that the **${}** token works on the variables that are used in **.foreach** and on true aliases.

Walking the Process List

The following example walks through the kernel-mode process list and displays the executable name for each entry in the list.

This example should be stored as a text file and executed with the **$$>< (Run Script File)** command. This command loads the whole file, replaces all carriage returns with semicolons, and executes the resulting block. This command enables you to write readable programs by using multiple lines and indentation, instead of having to squeeze the whole program onto a single line.

This example illustrates the following features:

- The **$t0**, **$t1**, and **$t2** pseudo-registers are used as variables in this program. The program also uses aliases named **Procc** and **$ImageName**.
- This program uses the MASM expression evaluator. However, the **@@c++( )** token appears one time. This token causes the program to use the C++ expression evaluator to parse the expression within the parentheses. This usage enables the program to use the C++ structure tokens directly.
- The **?** flag is used with the **r (Registers)** command. This flag assigns typed values to the pseudo-register **$t2**.

```
$$  Get process list LIST_ENTRY in $t0.
r $t0 = nt!PsActiveProcessHead

$$  Iterate over all processes in list.
.for (r $t1 = poi(@$t0);
      (@$t1 != 0) & (@$t1 != @$t0);
      r $t1 = poi(@$t1))
{
    r? $t2 = #CONTAINING_RECORD(@$t1, nt!_EPROCESS, ActiveProcessLinks);
    as /x Procc @$t2

 $$  Get image name into $ImageName.
 as /ma $ImageName @@c++(&@$t2->ImageFileName[0])

 .block
    {
        .echo ${$ImageName} at ${Procc}
    }

    ad $ImageName
    ad Procc
}
```

Walking the LDR_DATA_TABLE_ENTRY List

The following example walks through the user-mode LDR_DATA_TABLE_ENTRY list and displays the base address and full path of each list entry.

Like the preceding example, this program should be saved in a file and executed with the **$$>< (Run Script File)** command.

This example illustrates the following features:

- This program uses the MASM expression evaluator. However, in two places, the **@@c++( )** token appears. This token causes the program to use the C++ expression evaluator to parse the expression within the parentheses. This usage enables the program to use C++ structure tokens directly.
- The **?** flag is used with the **r (Registers)** command. This flag assigns typed values to the pseudo-registers **$t0** and **$t1**. In the body of the loop, **$t1** has the type **ntdll!_LDR_DATA_TABLE_ENTRY***, so the program can make direct member references.
- The user-named aliases **$Base** and **$Mod** are used in this program. The dollar signs reduce the possibility that these aliases have been used previously in the current debugger session. The dollar signs are not necessary. The **${/v: }** token interprets the alias literally, preventing it from being replaced if it was defined before the script is run. You can also use this token together with any block to prevent alias definitions before the block from being used.
- The **.block** token is used to add an extra alias replacement step. Alias replacement occurs one time for the whole script when it is loaded and one time when each block is entered. Without the **.block** token and its

braces, the **.echo** command does not receive the values of the **$Mod** and **$Base** aliases that are assigned in the previous lines.

```
$$ Get module list LIST_ENTRY in $t0.
r? $t0 = &@$peb->Ldr->InLoadOrderModuleList

$$ Iterate over all modules in list.
.for (r? $t1 = *(ntdll!_LDR_DATA_TABLE_ENTRY**)@$t0;
 (@$t1 != 0) & (@$t1 != @$t0);
      r? $t1 = (ntdll!_LDR_DATA_TABLE_ENTRY*)@$t1->InLoadOrderLinks.Flink)
{
    $$ Get base address in $Base.
 as /x ${/v:$Base} @@c++(@$t1->DllBase)

 $$ Get full name into $Mod.
 as /msu ${/v:$Mod} @@c++(&@$t1->FullDllName)

 .block
    {
        .echo ${$Mod} at ${$Base}
    }

    ad ${/v:$Base}
    ad ${/v:$Mod}
}
```

Using the WinDbg Graphical Interface

12/5/2018 • 2 minutes to read • Edit Online

This section includes the following topics:

Using Debugging Information Windows

Using Workspaces

Using the Toolbar and Status Bar

Using the Help Documentation

Using Debugging Information Windows

12/5/2018 • 2 minutes to read • Edit Online

WinDbg has ten kinds of debugging information windows. You can have only one instance of the following windows open at the same time: the Debugger Command window, the Watch window, the Locals window, the Registers window, the Calls window, the Disassembly window, the Processes and Threads window, and the Scratch Pad. In addition to these eight individual windows, WinDbg can display multiple Source windows and Memory windows at the same time.

This section describes the features that are common to all of these windows:

Opening a Window

Closing a Window

Configuring a Window

Moving Through a Window

Cutting and Pasting Text

Changing Text Properties

Positioning the Windows

Opening a Window

12/5/2018 • 2 minutes to read • Edit Online

When WinDbg begins a debugging session, the Debugger Command window automatically opens. The Disassembly window also automatically opens, unless you deselect Automatically Open Disassembly on the **Window** menu.

Whenever WinDbg discovers a source file that corresponds to the current program counter, WinDbg opens a Source window for that file. For other ways to open Source windows, see Source Path.

You can use the following menu commands, toolbar buttons, and shortcut keys to switch to these windows. That is, if a window is not open, it opens. If a window is open but inactive, it becomes active. If a window is docked and there is a floating window in front of it, the docked window becomes active but the floating window stays in front of the docked window.

WINDOW	MENU COMMAND	BUTTON	SHORTCUT KEYS
Debugger Command window	**View | Command**		ALT+1
Watch window	**View | Watch**		ALT+2
Locals window	**View | Locals**		ALT+3
Registers window	**View | Registers**		ALT+4
Memory window	**View | Memory**		ALT+5
Calls window	**View | Call Stack**		ALT+6
Disassembly window	**View | Disassembly**		ALT+7
Scratch Pad window	**View | Scratch Pad**		ALT+8
Processes and Threads window	**View | Processes and Threads**		ALT+9
Source window	Click File | Open Source File and then select a source file.		CTRL+O

You can also activate a window by selecting it from the list of open windows at the bottom of the **Window** menu.

Closing a Window

12/5/2018 • 2 minutes to read • Edit Online

To close a debugging information window, click the **Close** button in the upper-right corner of the window.

If you want to close the active window, you can also click **Close Current Window** on the **File** menu or press CTRL+F4.

You can close one of the debugging information windows by pressing ALT+SHIFT+*number*, where ALT+*number* are the shortcut keys that open this window. (For a list of the possible shortcut keys, see Opening a Window.)

To close all Source windows at the same time, click Close All Source Windows on the **Window** menu. This command will not close a Source window that has been designated as a tab-dock target. For more information on this setting, see the Source Window topic.

Configuring a Window

12/5/2018 • 2 minutes to read • Edit Online

Each debugging information window has a shortcut menu that you can access by right-clicking the title bar of the window or by clicking the icon near the upper-right corner of the title bar. You can use the commands on this shortcut menu to configure the window.

Many debugging information windows also have toolbars that contain buttons. Most of these buttons have features that are the same as commands on the shortcut menu.

Moving Through a Window

12/5/2018 • 2 minutes to read • Edit Online

There are several ways of moving through a debugging information window.

If a scrollbar appears in the window, you can use it to display more of the window.

In addition, some windows support the **Find**, **Go to Address**, or **Go to Line** commands. These commands only change the WinDbg display. They do not affect the execution of the target or any other debugger operations.

Find Command

You can use the **Find** command in the Debugger Command window or in a Source window.

When one of these windows is active, click **Find** on the **Edit** menu or press CTRL+F. The **Find** dialog box opens.

Enter the text that you want to find in the dialog box, and select **Up** or **Down** to determine the direction of your search. The search begins wherever the cursor is in the window. You can put the cursor at any point by using the mouse.

Select the **Match whole word only** check box if you want to search for a single whole word. (If you select this check box and you enter multiple words, this check box is ignored.) Select the **Match case** check box to perform a case-sensitive search.

If you close the **Find** dialog box, you can repeat the previous search in a forward direction by clicking **Find Next** on the **Edit** menu or pressing F3. You can repeat the search in a backward direction by pressing SHIFT+F3.

Go to Address Command

The **Go to Address** command searches for an address in the application that you are debugging. To use this option, click **Go to Address** on the **Edit** menu or press CTRL+G.

When the **View Code Offset** dialog box appears, enter the address that you want to search for. You can enter this address as an expression, such as a function, symbol, or integer memory address. If the address is ambiguous, a list appears with all of the ambiguous items.

After you click **OK**, the debugger moves the cursor to the beginning of the function or address in the Disassembly window or a Source window.

You can use the **Go to Address** command regardless of which debugging information window is open. If the debugger is in disassembly mode, the debugger finds the address that you are searching for in the Disassembly window. If the debugger is in source mode, the debugger tries to find the address in a Source window. If the debugger cannot find the address in a Source window, the debugger finds the address in the Disassembly window. If the needed window is not open, the debugger opens it.

Moving to a Specific Line

The **Go to Line** command searches for a line number in the active Source window. If the active window is not a Source window, you cannot use the **Go to Line** command.

To activate this option, click **Go to Line** on the **Edit** menu or press CTRL+L.

When the **Go to Line** dialog box appears, enter the line number that you want to find and then click **OK**. The debugger moves the cursor to that line. If the line number is larger than the last line in the file, the cursor moves to the end of the file.

Cutting and Pasting Text

12/5/2018 • 3 minutes to read • Edit Online

WinDbg uses many common methods of manipulating text and several methods that are less familiar.

Selecting Text

To select text in a Source window, in the Disassembly window, in either pane of the Debugger Command window, or in a dialog box, point to one end of the text, press and hold the left mouse button, and drag the pointer to the other end of the text.

To select all of the text in a window, you can also click Select All on the **Edit** menu or press CTRL+A.

In the Calls window, Watch window, Locals window, Registers window, and Memory window, you cannot select an arbitrary span of text, but you can select a whole line or cell. Click in the desired line or cell to select its text.

While you are entering text, press the DELETE and BACKSPACE keys to delete the text to the right or left of the cursor, respectively. If you select text, you can press these keys to delete the selection. If you select text and then type any characters, the new characters replace what you selected.

Copying Text

To copy text, select that text and then do one of the following:

- Press the right mouse button. (This method works only in some locations. For more information about how to use the right mouse button, see The Right Mouse Button.)
- Press CTRL+C.
- Press CTRL+INSERT.
- (Docked and tabbed windows only) Click **Copy** on the **Edit** menu.
- Click the **Copy (Ctrl+C)** button ( ) on the toolbar.

Cutting Text

To cut text and move it to the clipboard, select the text and then do one of the following:

- Press CTRL+X.
- Press SHIFT+DELETE.
- (Docked and tabbed windows only) Click **Cut** on the **Edit** menu.
- Click the **Cut (Ctrl+X)** button ( ) on the toolbar.

You can cut text from the bottom pane of the Debugger Command window, from the left column of the Watch window, and from any dialog box (that is, from any location that supports text entry).

Pasting Text

To paste text from the clipboard, put the cursor where you want to insert the text (or select the text that you want to replace) and then do one of the following:

- Press the right mouse button. (This method works only in some locations, and you cannot use this method to replace text. For more inormation about how to use this method, see The Right Mouse Button.)
- Press CTRL+V.

- Press SHIFT+INSERT.
- (Docked and tabbed windows only) Click **Paste** on the **Edit** menu.
- Click the **Paste (Ctrl+V)** button ( ) on the tooblar.

You can paste text into the bottom pane of the Debugger Command window, into the left column of the Watch window, and into any dialog box (that is, into any location that supports text entry).

Right Mouse Button

The right mouse button has several effects that can make copying and pasting much quicker:

- If you select text in either pane of the Debugger Command window, in the Scratch Pad, in the Disassembly window, or in any Source window, and then you press the right mouse button, the text is copied to the clipboard. However, if **QuickEdit Mode** has been deselected in View | Options, right-clicking in these locations will pop up the menu most relevant to the current location.
- If you put the cursor (without selecting any text) in either pane of the Debugger Command window, in the Scratch Pad, or in the text entry space of the Watch window, and then you press the right mouse button, the contents of the clipboard are pasted into the window. However, if **QuickEdit Mode** has been deselected in View | Options, right-clicking in these locations will pop up the menu most relevant to the current location.
- If you put the cursor in any box and then press the right mouse button, a menu with **Undo**, **Cut**, **Copy**, **Paste**, and **Select All** options appears. You can choose any of these options.

Changing Text Properties

12/5/2018 • 2 minutes to read • Edit Online

You can customize the font that is used in all of the debugging information windows. You can also change the tab width that is used in the Source windows.

Setting the Font, Font Style, and Font Size

All debugging information windows share the same font. To change this font, click **Font** on the **View** menu, or click the **Font** button ( ) on the toolbar.

In the **Font** dialog box, select the font, style, and size from the appropriate lists, and then click **OK**. All of the available fonts are fixed-pitch, because these kinds of fonts are the most useful for viewing code.

Setting the Tab Width

To change the tab width settings, click **Options** on the **View** menu or click the **Options** button ( ) on the toolbar.

The **Options** dialog box then appears. In the **Tab width** box, enter the number of spaces that the tab width should be equivalent to, and then click **OK**.

The tab settings affect the display of the code in any Source window.

Positioning the Windows

12/5/2018 • 2 minutes to read • Edit Online

Each debugging information window can be *floating* or *docked*.

You can position *floating windows* separately from other windows. *Docked windows* are connected to the WinDbg window or to an independent dock. Each docked window can occupy a unique position in its dock or can be combined with other docked windows in a *tabbed collection*.

This section includes the following topics:

Debugging with Floating and Docked Windows

Docking a Window

Tabbing a Window

Undocking a Window

Creating a New Dock

Resizing and Moving Windows

Arranging Windows

Debugging with Floating and Docked Windows

12/5/2018 • 2 minutes to read • Edit Online

The features that are available in a debugging information window are not affected by whether the window is floating, docked, or docked in a tabbed collection.

Overview of the Window Configuration

A floating window is not connected to the WinDbg window or any other dock. Floating windows always appear directly in front of the WinDbg window.

A docked window occupies a fixed position within the WinDbg window or in a separate dock.

When two or more docked windows are tabbed together, they occupy the same position within the frame. You can see only one of these tabbed windows at one time. At the bottom of each tabbed window collection is a set of tabs. The selected tab indicates which window in the collection is visible.

Making a Window Active

You can make any window active, regardless of its position. When a floating window is active, it appears in the foreground. When a window that is inside an additional dock is active, that dock appears in the foreground. When a docked window within the WinDbg window is active, one or more floating windows might still obscure the docked window.

To make a floating window or a docked window active, click its title bar. To make a docked window in a tabbed collection active, click its tab.

You can also make a window active by using the WinDbg menu or toolbar. You can activate any window by clicking the window name at the bottom of the **Window** menu. You can also activate any window (other than a Memory window or a Source window) by clicking its name on the **View** menu or clicking its toolbar button.

Press CTRL+TAB to switch between debugging information windows. By pressing these keys repeatedly, you can scan through all of the windows, regardless of whether they are floating, docked by themselves, or part of a tabbed collection of docked windows. When you release the CTRL key, the window that you are currently viewing becomes active.

The ALT+TAB shortcut keys are the standard Microsoft Windows shortcut keys to switch between the windows on the desktop. You can use these shortcut keys to switch between the WinDbg window and any additional docks that you have created. You can also make a dock active by clicking its button in the Windows taskbar.

Docking a Window

12/5/2018 • 3 minutes to read • Edit Online

To dock a floating window, do one of the following:

- Double-click the window's title bar.
- Open the shortcut menu by right-clicking the window's title bar or clicking the window's icon in the upper-right corner, and then click **Dock**.
- In the WinDbg window, on the **Window** menu, click **Dock All**. This command docks all of the windows except those that have the **Always floating** option selected on their individual shortcut menus.
- Drag the window to a docking location. This action causes the window to dock unless **Always floating** is selected on the shortcut menu for that window, or unless you press and hold the ALT key as you begin dragging the window.

When you dock a window by any method other than dragging it, WinDbg automatically positions the docked window. If the window has never been docked before, WinDbg moves the window to a new untabbed location within the WinDbg window. If the window has been docked before, WinDbg returns the window to its most recent docking location, which might be tabbed or untabbed.

When you dock a window by dragging it, you can control its destination position. As you drag the window, you will see a semi-transparent outline of the window appear. This outline shows where the window will be docked if you release the mouse button at that point. The following rules determine where a dragged window is docked:

- If you drag the mouse pointer over the WinDbg window when the window is empty or over an empty dock, and then you release the mouse button, the dragged window is docked in that location and completely fills the frame or dock.
- If you drag the mouse pointer over the left, right, top, or bottom portion of an already-docked window and then you release the mouse button, the dragged window is docked to the left, right, top, or bottom of the already-docked window, respectively.
- When you drag the mouse pointer over a floating window (including the original position of the window that you are dragging), no docking occurs. This exception means that you might have to drag other windows out of the way (or drag the current window two times) before you can move the window to where you want it.
- If you drag the mouse pointer to a position that is not inside the WinDbg frame or any other dock and then you release the mouse button, the dragged window remains floating.

All of the preceding rules apply to the mouse pointer location itself. They do not depend on where you originally clicked within the title bar of the window that you are dragging.

Re-docking

If you let WinDbg automatically dock a floating window that was previously docked, WinDbg tries to put the window in the same docking position that it previously occupied. Also, if you load a workspace, WinDbg tries to restore all of the debugging information windows to their previous positions, whether docked or floating.

However, multiple instances of Memory windows and Source windows are not distinguished when the docking position is saved. For example, if you combine the Locals window together with a Memory window in a tabbed collection, and this state is saved and later restored, the Locals window joins a Memory window in a tabbed collection, but it might not be the same Memory window as before.

If you load a workspace that includes one or more Source windows when the source files are inaccessible, those Source windows are not reopened. When this situation occurs, other windows that were tabbed together with those windows might return to the floating state. If you want to keep all of your Source windows tabbed together, you should include at least one source file that is always present, or include an additional window in the tabbed collection.

Tabbing a Window

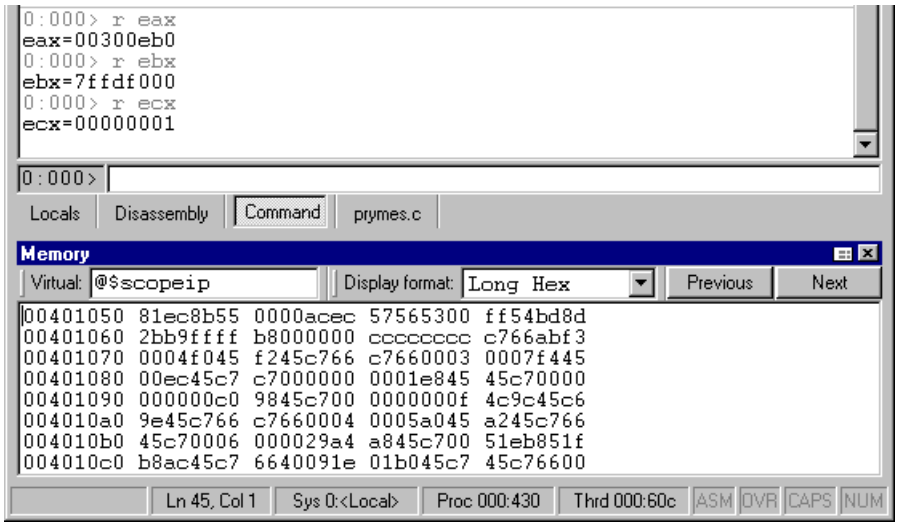
12/5/2018 • 2 minutes to read • Edit Online

To tab a floating or docked window, drag it on top of another docked window. Drag the window with the mouse pointer over the center of an already-docked window, and then release the mouse button. The dragged window joins the already-docked window as a tabbed window collection.

All Source windows can be grouped automatically into a tabbed collection by selecting one Source window and designating it as the tab-dock target for all Source windows by choosing the **Set as tab-dock target for window type** option in its short-cut menu. Once this is done, all future Source windows that are opened will automatically be included in a tabbed collection with this first Source window. The Source window marked as the tab-dock target will not be closed when the **Window | Close All Source Windows** menu command is selected. Thus you can set up a placeholder window for the Source windows without worrying that it will be closed when you don't want it to be. The same process also works for Memory windows.

Note If you want a window to join another window in a tabbed window collection, watch the outline of the window that moves as you drag the window. When this outline completely surrounds the window that you want to join, release the mouse button.

A set of tabs always controls the window immediately above the tabs. In the following illustration, the Debugger Command window is selected and is visible above the tabs.



Undocking a Window

12/5/2018 • 2 minutes to read • Edit Online

To undock a window and make it a floating window, do one of the following:

- Double-click the window's title bar.
- Open the shortcut menu by right-clicking the window's title bar, right-clicking the window's tab if it is part of a tabbed collection, or clicking the window's icon in the upper-right corner, and then click **Undock**.
- In the WinDbg window, on the **Window** menu, click **Undock All**. This command changes all of the docked windows into floating windows.

When you undock a window by one of the preceding methods, the window returns to its previous undocked position.

You can also drag a docked window by clicking its title bar. This action enables you to move the window to a different docked position or undock it. (Dragging a docked window to a new position works exactly like dragging a floating window to a new position. For more information about dragging a floating window to a new position, see Docking a Window.)

When you try to undock or drag a tabbed window by any of these methods, only the active window in the tabbed collection is moved.

Creating a New Dock

12/5/2018 • 2 minutes to read • Edit Online

When WinDbg is started, the WinDbg window is the only possible docking location.

To create a new dock, on the **Window** menu, click **Open Dock**. This dock is an independent window that you can maximize, minimize, or drag like any other window on the desktop.

A new dock can also be created by using the **Move to new dock** option on the shortcut menu for any already open window. This selection will close the window and open it in a new dock.

You can create as many docks as you want.

To close a dock and any debugging information windows that are currently docked there, click the **Close** button in the upper-right corner of the dock.

Resizing and Moving a Window

12/5/2018 • 2 minutes to read • Edit Online

Floating windows are always associated with the WinDbg window. If you minimize WinDbg, all floating windows are minimized. And if you restore WinDbg, all floating windows are restored. You can never put a floating window behind the WinDbg window.

Each floating window moves independently from each other and from the WinDbg window, unless you have selected **Move with frame** on the window's shortcut menu.

A docked window occupies a fixed position within the WinDbg frame. If you resize WinDbg, all of the docked windows are automatically scaled to the new size. The same situation applies to windows that have been docked in a separate dock.

If you move the mouse pointer to the border between two docked windows, the mouse pointer becomes an arrow. By dragging this arrow, you can resize the two adjacent windows and leave them in the docked state.

The WinDbg window is always filled with docked windows. There is never any empty area in the window unless there are no windows docked. The same situation applies to independent docks.

Arranging Windows

12/5/2018 • 2 minutes to read • Edit Online

One useful window arrangement is to combine all of your Source windows into a single tabbed collection. The easiest way to do this is by marking your first source window as the tab-dock target for all Source windows by selecting the **Set as tab-dock target for window type** option in the Source window's short-cut menu. Once this is done, all future Source windows that are opened will automatically be included in a tabbed collection with this first Source window. The Source window marked as the tab-dock target will not be closed when the **Window | Close All Source Windows** menu command is selected. Thus you can set up a placeholder window for the Source windows that will only be closed when you want it to be.

This collection can occupy half of the WinDbg window or you can put it in a separate dock.

If you want each debugging information window to be completely separate, you can create one dock for each window. This arrangement enables you to minimize or maximize each window separately.

If you want all of your windows to be floating, you should select **Always floating** on each window's shortcut menu so that you can drag each window independently to any location.

Alternatively, you can use the MDI Emulation command on the **Window** menu. This command makes all of the windows floating windows and constrains them within the frame window. This behavior emulates the behavior of WinDbg before the introduction of docking mode.

If you are using dual monitors, you can put the WinDbg window in one monitor and an extra dock in the other.

Some standard window arrangements for various debugging scenarios are included in the Debugging Tools for Windows package. For details on these arrangements, see Using and Customizing WinDbg Themes.

Using Workspaces

12/5/2018 • 2 minutes to read • Edit Online

When you exit WinDbg, it saves your session configuration in a *workspace*. A workspace enables you to easily preserve your settings from one session to another. You can also save or clear the workspaces manually, or even use a workspace to save a debugging session that is still in progress.

This section includes the following topics:

Creating and Opening a Workspace

Workspace Contents

Using and Customizing WinDbg Themes

Creating and Opening a Workspace

12/5/2018 • 3 minutes to read • Edit Online

WinDbg has two kinds of workspaces: *default workspaces* and *named workspaces*.

Default Workspaces

WinDbg has several different kinds of default workspaces:

- The *base workspace* is used when WinDbg is in a dormant state.
- The *default user-mode workspace* is used when you are attaching to a user-mode process (by using the **-pcommand-line option** or by using the File | Attach to a Process command).
- The *remote default workspace* is used when you are connecting to a debugging server.
- The *default kernel-mode workspace* is used when WinDbg begins a kernel-mode debugging session.
- The *processor-specific workspace* is used during kernel-mode debugging after WinDbg attaches to the target computer. There are separate processor-specific workspaces for x86-based and x64-based processors.

When WinDbg creates a user-mode process for debugging, a workspace is created for that executable file. Each created executable file has its own workspace.

When WinDbg analyzes a dump file, a workspace is created for that dump file analysis session. Each dump file has its own workspace.

When you begin a debugging session, the appropriate workspace is loaded. When you end a debugging session or exit WinDbg, a dialog box is displayed and asks you if you want to save the changes that you have made to the current workspace. If you start WinDbg with the **-QYcommand-line option**, this dialog box does not appear, and workspaces are automatically saved. Also, if you start WinDbg by the **-Q** command-line option, this dialog box does not appear, and no changes are saved.

Workspaces load in a cumulative manner. The base workspace is always loaded first. When you begin a particular debugging action, the appropriate workspace is loaded. So most debugging is completed after two workspaces have been loaded. Kernel-mode debugging is completed after three workspaces have been loaded (the base workspace, the default kernel-mode workspace, and the processor-specific workspace).

For greatest efficiency, you should save settings in lower-level workspaces if you want them to apply to all of your WinDbg work.

Note The layout of the debugging information windows is one exception to the cumulative behavior of workspaces. The position, docking status, and size of each window are determined by only the most recent workspace that you opened. This behavior includes the contents of the Watch window and the locations that you viewed in each Memory window. The command history in the Debugger Command window is not cleared when a new workspace is opened, but all other window states are reset.

To access the base workspace, start WinDbg with no target, or click Stop Debugging on the **Debug** menu after your session is complete. You can then make any edits that are allowed in the base workspace.

Named Workspaces

You can also give workspaces names and then save or load them individually. After you load a named workspace, all automatic loading and saving of default workspaces is disabled.

Named workspaces contain some additional information that default workspaces do not. For more information about this additional information, see Workspace Contents.

Opening, Saving, and Clearing Workspaces

To control workspaces, you can do the following:

- Open and load a named workspace by using the **-W command-line option**.
- Open and load a workspace from a file by using the **-WF command-line option**.
- Disable all automatic workspace loading by using the **-WX command-line option**. Only explicit workspace commands cause workspaces to be saved or loaded.
- Open and load a named workspace by clicking Open Workspace on the **File** menu or pressing CTRL+W.
- Save the current default workspace or the current named workspace by clicking Save Workspace on the **File** menu.
- Assign a name to the current workspace and save it by clicking Save Workspace As on the **File** menu.
- Delete specific items and settings from the current workspace by clicking Clear Workspace on the **File** menu.
- Delete workspaces by clicking Delete Workspaces on the **File** menu.
- Open and load a workspace from a file by clicking Open Workspace in File on the **File** menu.
- Save a workspace to a file by clicking Save Workspace to File on the **File** menu.

Workspace Contents

12/5/2018 • 3 minutes to read • Edit Online

Each workspace preserves the following information about the current debugging session. This information is applied cumulatively, starting with the base workspace and ending with the most recently-loaded workspace.

- All break and handling information for exceptions and events. For more information about the break and handling information, see Breakpoints in Workspaces.
- All open source files. If a source file is not found, an error message appears. You can close these error messages individually or by using the Window | Close All Error Windows command.
- All user-defined aliases.

Each workspace preserves the following information about the debugger configuration settings. This information is applied cumulatively, starting with the base workspace and ending with the most recently-loaded workspace.

- The symbol path.
- The executable image path.
- The source path. (In remote debugging, the main source path and the local source path are saved.)
- The current source options that were set with **l+, l- (Set Source Options)**.
- Log file settings.
- The COM or 1394 kernel connection settings, if the connection was started by using the graphical interface.
- The most recent paths in each **Open** dialog box (except for the workspace file and text file paths, which are not saved).
- The current **.enable_unicode**, **.force_radix_output**, and **.enable_long_status** settings.

All default workspaces and named workspaces preserve the following information about the WinDbg graphical interface. This information is loaded cumulatively, starting with the base workspace and ending with the most recently-loaded workspace.

- The title of the WinDbg window
- The Automatically Open Disassembly setting
- The default font

All default workspaces and named workspaces preserve the following information about the WinDbg graphical interface. This information is not applied cumulatively. It depends only on the most recently-loaded workspace.

- The size and position of the WinDbg window on the desktop.
- Which debugging information windows are open.
- The size and position of each open window, including the window's size, its floating or docked status, whether it is tabbed with other windows, and all of the related settings in its shortcut menu.
- The location of the pane boundary in the Debugger Command window and the word wrap setting in that window.
- Whether the toolbar and status bar, and the individual toolbars on each debugging information window, are

visible.

- The customization of the Registers window.
- The flags in the Calls window, Locals window, and Watch window.
- The items that were viewed in the Watch window.
- The cursor location in each Source window.

Named Workspaces

Named workspaces contain additional information that is not stored in default workspaces.

This additional information includes information about the current session state. When a named workspace is saved, the current session is saved. If this workspace is later opened, this session is automatically restarted.

You can start only kernel debugging, dump file debugging, and debugging of spawned user-mode processes in this manner. Remote sessions and user-mode processes that the debugger attached to do not have this session information saved in their workspaces.

You cannot open this kind of named workspace if another session is already active.

Debugging Clients and Workspaces

When you use WinDbg as a debugging client, its workspace saves only values that you set through the graphical interface. Changes that you make through the Debugger Command window are not saved. (This restriction guarantees that only changes that the local client made are reflected, because the Debugger Command window accepts input from all clients and the debugging server.) For more information, see Controlling a Remote Debugging Session.

Breakpoints in Workspaces

In addition, breakpoint information is saved in workspaces, including the break address and status. Breakpoints that are active when a session ends are active when the next session is started. However, some of these breakpoints might be unresolved if the proper modules have not yet been loaded.

Breakpoints that you specify by a symbol expression, by a line number, by a numeric address, or by using the mouse in a Source window are all saved in workspaces. Breakpoints that you specify by using the mouse in a Disassembly or Calls window are not saved in workspaces.

If you are debugging multiple user-mode processes, only breakpoints that are associated with process zero are saved.

Using and Customizing WinDbg Themes

12/5/2018 • 2 minutes to read • Edit Online

A theme is a preconfigured WinDbg workspace that contains a useful configuration of debugging information windows.

Any theme can be saved as your base workspace. Themes in the Debugging Tools for Windows package are provided as a set of registry files (with the .reg extension). As you accumulate more debugging sessions, various default workspaces are automatically set up. These default workspaces use the base workspace as a starting point. For more information on default workspaces, see Creating and Opening a Workspace.

For the best user experience, we recommend that you follow the instructions in this topic before starting your debugging session.

This section includes the following topics:

Loading a Theme

Customizing a Theme

Using Themes Provided in Debugging Tools for Windows

Loading a Theme

12/5/2018 • 2 minutes to read • Edit Online

Before loading a theme, we recommend that you clear all of your workspace data. This can be done in three ways:

By using the WinDbg user-interface. Under the **File** menu, select **Clear Workspace...** Select **Clear All** in the pop-up window and then click **OK**.

By deleting the registry key under **HKCU\Software\Microsoft\Windbg\Workspaces**.

By running the command **reg delete HKCU\Software\Microsoft\Windbg**.

After all of your workspace data has been cleared, run one of the themes. These are stored as .reg files in the Themes directory of your Debugging Tools for Windows installation. Running a theme imports its settings into the registry, redefining your base workspace.

After you have loaded a theme, you may alter it to more closely suit your preferences. For more details on some common options, see Customizing a Theme.

Customizing a Theme

12/5/2018 • 2 minutes to read • Edit Online

Before customizing a theme, it must first be loaded. See Loading a Theme for details.

After the theme is loaded, start WinDbg with no command-line parameters. This opens the base workspace. There are two common areas of focus for customizing a theme: setting paths and adjusting window position.

After you have completed any wanted adjustments, exit WinDbg and save your workspace by selecting **Save Workspace** from the **File** menu. If you want to save your new settings to a .reg file, open Regedit and export the registry key under **HKCU\Software\Microsoft\Windbg\Workspaces** to a .reg file.

Setting Paths

By setting the appropriate paths, you can ensure that WinDbg can locate all of the files that it needs to debug effectively. There are three main paths to set: the symbol path, the source path, and the executable image path.

Here are examples of how to set the symbol and source path. The executable image path is typically the same as your symbol path.

To set your symbol path:

```
SRV*c:\MySymCache*\\CompanySymbolServer\Symbols;SRV*c:\WinSymCache*https://msdl.microsoft.com/download/symbols
```

To set your source path:

```
SRV*;d:\MySourceRoot
```

Adjusting Window Position

Before using your theme, you should adjust the window positioning so that WinDbg handles your source files correctly. This ensures that Source windows knows where to dock.

Begin by opening a Source window in WinDbg. Tab-dock this window with the placeholder set aside for your Source windows. In order for the proper relationship to be made, the placeholder window must be the uppermost window in the dock before you perform this tab-docking operation. Now close the source window but not the placeholder window.

Because debugging information windows "remember" their last docking operation, each source window's last docking operation is associated with one of the placeholder windows after you have performed this procedure. Because of this memory attribute, you should not close any of your placeholder windows. Further, if you choose to change the theme's configuration, any window you reposition in a dock should always be tab-docked with a placeholder file.

The sample themes included with the Debugging Tools for Windows were created using the following actions:

Place and position the placeholder*.c files into the dock.

Tab-dock every window type above the wanted placeholder window.

For further information about adjusting window position in WinDbg, see Positioning the Windows.

Using Themes Provided in Debugging Tools for Windows

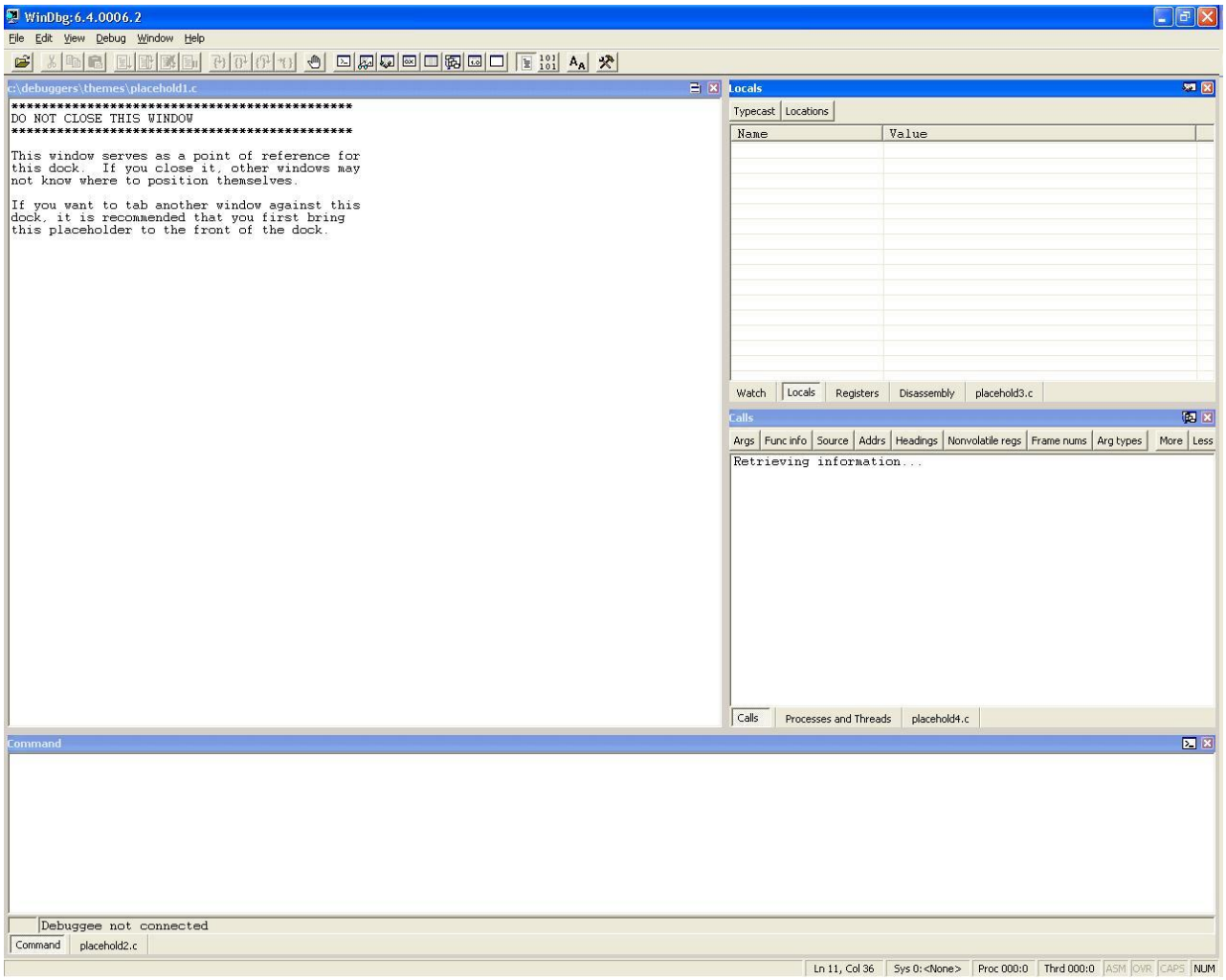
12/5/2018 • 3 minutes to read • Edit Online

This topic shows screen shots of the configurations from each of the four themes provided in Debugging Tools for Windows. Those themes are Standard.reg, Standardvs.reg, Srcdisassembly.reg, and Multimon.reg.

Standard.reg

The Standard.reg theme can be used for most debugging purposes. In this arrangement, the lower third of the WinDbg window is taken by the Debugger Command window. The upper two-thirds is divided roughly in half. The left half is taken up by a placeholder window that indicates where the Source windows open in a tabbed collection. The right half is further divided into halves vertically. The upper half contains a tabbed collection that includes the Watch, Locals, Registers, and Disassembly windows. The lower half contains a tabbed collection that includes the Calls and Processes and Threads windows.

In each docking location, a placeholder window is also included as a point of reference for the other windows. The placeholder windows should not be closed because closing them may change the configuration of the windows. All of the windows in this arrangement are docked.The following screen shot shows the Standard.reg theme.

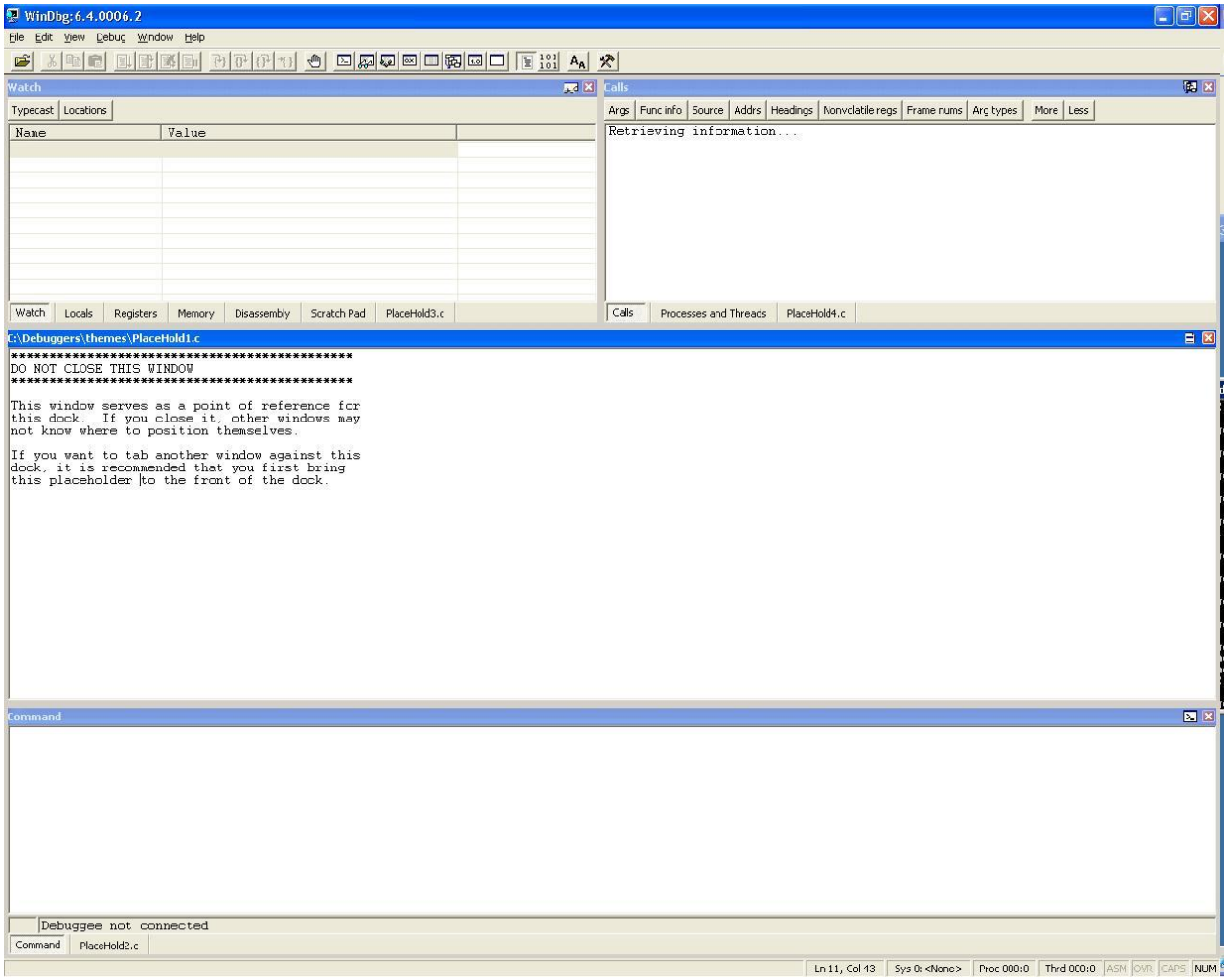


Standardvs.reg

The Standardvs.reg theme can be used for most debugging purposes, but is more similar in layout to Visual Studio. In this arrangement, the WinDbg window is divided horizontally into thirds. The upper third is further

divided vertically into halves. The left half of the upper third contains a tabbed collection that includes the Watch, Locals, Registers, Memory, Disassembly, and Scratchpad windows. The right half of the upper third contains a tabbed collection that includes the Calls and Processes and Threads windows. The lower third of the WinDbg window is taken by the Debugger Command window. The middle third is filled by a placeholder window that indicates where the Source windows are opened in a tabbed collection.

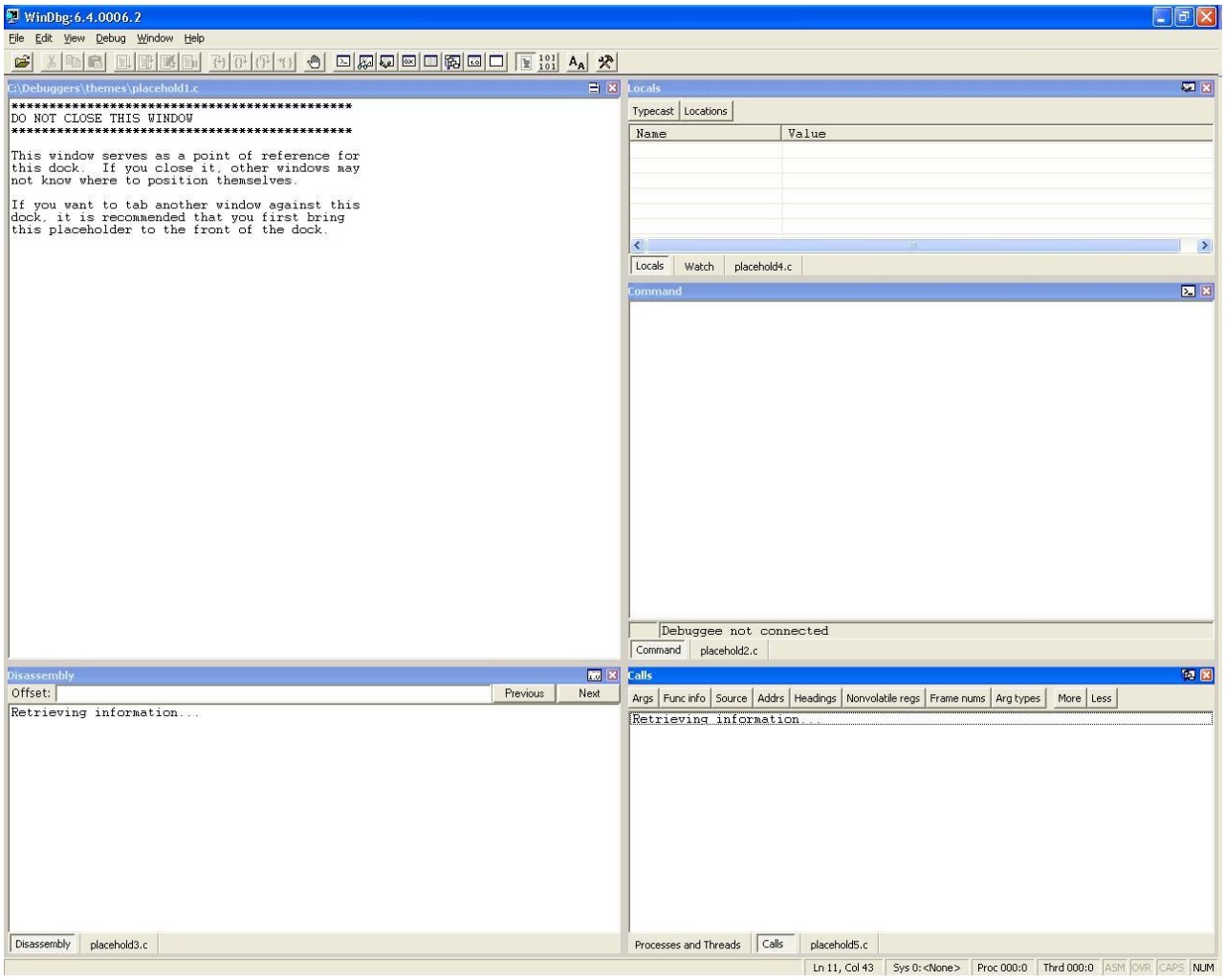
In each docking location, a placeholder window is also included as a point of reference for the other windows. The placeholder windows should not be closed because closing them may change the configuration of the windows. All of the windows in this arrangement are docked. The following screen shot shows the Standardvs.reg theme.



Srcdisassembly.reg

The Srcdisassembly.reg theme includes a Disassembly window, for debugging in assembly mode. In this arrangement, the WinDbg window is divided in half vertically, and each half formed is further divided into thirds horizontally. On the right half, the upper third is a tabbed collection of the Locals and Watch windows, the middle third is the Debugger Command window, and the lower third is a tabbed collection of the Processes and Threads and Calls windows. On the left half, the upper two-thirds are taken by a placeholder window that indicates where the Source windows opens in a tabbed collection; the lower third is taken up by the Disassembly window.

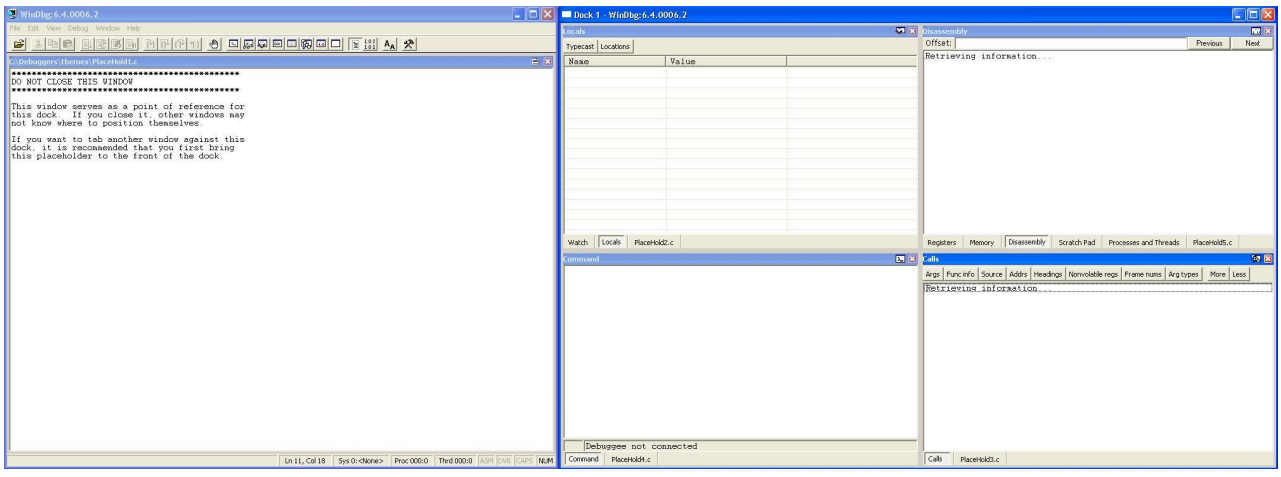
In each docking location, a placeholder window is also included as a point of reference for the other windows. The placeholder windows should not be closed because closing them may change the configuration of the windows. All of the windows in this arrangement are docked. The following screen shot shows the Srcdisassembly.reg theme.



Multimon.reg

The Multimon.reg theme is set up for debugging with multiple monitors. In this arrangement, a new dock is created so that the WinDbg window can be viewed on one monitor and the new dock can be viewed on the other monitor. The WinDbg window is filled by a placeholder window that indicates where the Source windows open in a tabbed collection. The new dock is divided into fourths. The upper left contains a tabbed collection that includes the Watch and Locals windows. The upper right contains a tabbed collection that includes the Registers, Memory, Disassembly, Scratchpad, and Processes and Threads windows. The lower left contains the Debugger Command window. The lower right contains the Calls window.

In each docking location, a placeholder window is also included as a point of reference for the other windows. The placeholder windows should not be closed because closing them may change the configuration of the windows. All of the windows in this arrangement are docked. The following screen shot shows the Multimon.reg theme.



Using the Toolbar and Status Bar

12/5/2018 • 2 minutes to read • Edit Online

The *toolbar* appears underneath the menu bar, near the top of the WinDbg window. The*status bar* appears at the bottom of the WinDbg window.

Using the Toolbar

The following screen shot shows the WinDbg toolbar.

The toolbar buttons have various effects. Most of them are equivalent to menu commands. To execute the command that is associated with a toolbar button, click the toolbar button. When you cannot use a button, it appears unavailable.

For more information about each button, see Toolbar Buttons.

Using the Status Bar

The following screen shot shows the WinDbg status bar.

The following table describes the sections of the WinDbg status bar.

SECTION	DESCRIPTION
Message	Displays messages from the debugger.
Ln, Col	Displays the line number and column number at the cursor in the active Source window.
Sys	Shows the internal decimal number of the system that you are debugging, followed by its computer name (or **<Local>** if it is the same as the computer as the debugger is running on).
Proc	Shows the internal decimal number of the process that you are debugging, followed by its hexadecimal process ID.
Thrd	Shows the internal decimal number of the thread that you are debugging, followed by its hexadecimal thread ID.
ASM	Indicates that WinDbg is in assembly mode. If ASM is unavailable, WinDbg is in source mode.
OVR	Indicates that overtype mode is active. If OVR is unavailable, insert mode is active.

SECTION	DESCRIPTION
CAPS	Indicates that CAPS LOCK is on.
NUM	Indicates that NUM LOCK is on.

Hiding the Toolbar or Status Bar

To display or hide the toolbar, select or clear Toolbar on the **View** menu. To display or hide the status bar, select or clear Status Bar on the **View** menu.

If you hide the toolbar or the status bar, you have more space for debugging information windows in the WinDbg display area.

Setting the Window Title

You can change the title of the WinDbg window by using the **.wtitle (Set Window Title)** command.

Using the Help Documentation

9/25/2019 • 3 minutes to read • Edit Online

The WinDbg Help documentation that you are reading is the documentation for the Debugging Tools for Windows package.

This documentation uses the viewer that is supplied with Microsoft HTML Help (hh.exe). This viewer provides a table of contents and index and enables you to search through the documentation, mark your favorite or frequently used topics, and print one or more topics.

The following sections in this topic describe the features of and how to use the Help documentation.

Contents Tab

The **Contents** tab provides an expandable view of the documentation's table of contents.

Click the plus sign (**+**) next to a node or double-click the node's title to expand or contract the table of contents under that node.

Index Tab

The **Index** tab displays the complete index of terms that are used in this documentation. You can enter the beginning of a term or use the scrollbar to look through this index.

Search Tab

In the **Search** tab, you can search for any word or phrase that is contained in the documentation. You can search all text or limit your search to topic titles.

To search for phrases that contain more than one word, enclose the phrase in quotation marks. You can connect multiple words and phrases with the AND, OR, and NOT operators. The default operator is AND.

Note that "AND NOT" is invalid. To search for topics that contain *x* and not *y*, use "*x* NOT *y*". You can also use NEAR in searches.

Wildcard characters are also permitted. Use a question mark (**?**) to represent any single character and an asterisk (*****) to represent zero or more characters. However, you cannot use wildcard characters within quoted strings.

All letters and numbers are treated literally, but some symbols are not permitted in searches.

A search returns a list of all topics that match the specified criteria. If you select the **Search previous results** box, you can then search these results for more terms.

Favorites Tab

In the **Favorites** tab, you can save the titles of commonly-visited topics. While you are viewing a topic, click the **Favorites** tab and then click the **Add** button.

To rename a page on your favorites list, click its name two times slowly, and retype the name. To view a topic on the Favorites list, double-click its name or click it one time and then click **Display**. To remove a topic from the favorites list, click it one time and then click **Remove**.

Printing Topics

To print one or more topics, click a topic in the **Contents** tab, and then click the **Print** button on the toolbar. You will be asked whether you want to print only the selected topic or that topic and all of its subtopics.

You can also print the topic that you are viewing by right-clicking within the view window, and clicking **Print** on the shortcut menu. However, this method does not enable you to print subtopics.

Searching Within a Topic

To search for a text string within the topic that you are viewing, press CTRL+F, or click **Find in this Topic** on the **Edit** menu.

Accessing the Help Documentation

To open this Help documentation, do one of the following:

- Click **Start**, point to **All Programs**, point to **Debugging Tools for Windows**, and then click **Debugging Help**.
- Open Windows Explorer, locate the Debugger.chm file, and then double-click it.
- At a command prompt, browse to the folder that contains Debugger.chm and run **hh debugger.chm**.
- In any Windows debugger, use the **.hh (Open HTML Help File)** command.
- In WinDbg, click **Contents** on the **Help** menu. This command open the Help documentation and opens the **Contents** tab.
- In WinDbg, click **Index** on the **Help** menu. This command open the Help documentation and opens the **Index** tab.
- In WinDbg, click **Search** on the **Help** menu. This command opens the Help documentation and opens the **Search** tab.
- Many dialog boxes in WinDbg have **Help** buttons. Click **Help** to open this documentation and open the relevant page.

Using Debugger Extensions

12/5/2018 • 2 minutes to read • Edit Online

Visual Studio, WinDbg, KD, and CDB all allow the use of debugger extension commands. These extensions give these three Microsoft debuggers a great degree of power and flexibility.

Debugger extension commands are used much like the standard debugger commands. However, while the built-in debugger commands are part of the debugger binaries themselves, debugger extension commands are exposed by DLLs distinct from the debugger.

This allows you to write new debugger commands which are tailored to your specific need. In addition, a number of debugger extension DLLs are shipped with the debugging tools themselves.

This section includes:

Loading Debugger Extension DLLs

Using Debugger Extension Commands

Writing New Debugger Extensions

Loading Debugger Extension DLLs

12/5/2018 • 2 minutes to read • Edit Online

There are several methods of loading debugger extension DLLs, as well as controlling the default debugger extension DLL and the default debugger extension path:

- (Before starting the debugger) Use the _NT_DEBUGGER_EXTENSION_PATH environment variable to set the default path for extension DLLs. This can be a number of directory paths, separated by semicolons.
- Use the **.load (Load Extension DLL)** command to load a new DLL.
- Use the **.unload (Unload Extension DLL)** command to unload a DLL.
- Use the **.unloadall (Unload All Extension DLLs)** command to unload all debugger extensions.
- (Before starting the debugger; CDB only) Use the tools.ini file to set the default extension DLL.
- (Before starting the debugger) Use the **-a** command-line option to set the default extension DLL.
- Use the **.extpath (Set Extension Path)** command to set the extension DLL search path.
- Use the **.setdll (Set Default Extension DLL)** command to set the default extension DLL.
- Use the **.chain (List Debugger Extensions)** command to display all loaded debugger extension modules, in their default search order.

You can also load an extension DLL simply by using the full **!***module***.***extension* syntax the first time you issue a command from that module. See Using Debugger Extension Commands for details.

The extension DLLs that you are using must match the operating system of the target computer. The extension DLLs that ship with the Debugging Tools for Windows package are each placed in a different subdirectory of the installation directory:

- The winxp directory contains extensions that can be used with Windows XP and later versions of Windows.
- The winext directory contains extensions that can be used with any version of Windows. The dbghelp.dll module, located in the base directory of Debugging Tools for Windows, also contains extensions of this type.

If you write your own debugger extensions, you can place them in any directory. However, it is advised that you place them in a new directory and add that directory to the debugger extension path.

There can be as many as 32 extension DLLs loaded.

Using Debugger Extension Commands

1/3/2019 • 2 minutes to read • Edit Online

The use of debugger extension commands is very similar to the use of debugger commands. The command is typed in the Debugger Command window, producing either output in this window or a change in the target application or target computer.

An actual debugger extension command is an entry point in a DLL called by the debugger.

Debugger extensions are invoked by the following syntax:

![*module***.**]*extension* [*arguments*]

The module name should not be followed with the .dll file name extension. If *module* includes a full path, the default string size limit is 255 characters.

If the module has not already been loaded, it will be loaded into the debugger using a call to **LoadLibrary**(*module*). After the debugger has loaded the extension library, it calls the **GetProcAddress** function to locate the extension name in the extension module. The extension name is case-sensitive and must be entered exactly as it appears in the extension module's .def file. If the extension address is found, the extension is called.

Search Order

If the module name is not specified, the debugger will search the loaded extension modules for this export.

The default search order is as follows:

1. The extension modules that work with all operating systems and in both modes: Dbghelp.dll and winext\ext.dll.
2. The extension module that works in all modes but is operating-system-specific. For Windows XP and later versions of Windows, this is winxp\exts.dll. There is no corresponding module for Windows 2000.
3. The extension module that works with all operating systems but is mode-specific. For kernel mode, this is winext\kext.dll. For user mode, this is winext\uext.dll.
4. The extension module that is both operating-system-specific and mode-specific. The following table specifies this module.

USER MODE	KERNEL MODE
winxp \ ntsdexts.dll	winxp \ kdexts.dll

When an extension module is unloaded, it is removed from the search chain. When an extension module is loaded, it is added to the beginning of the search order. The **.setdll (Set Default Extension DLL)** command can be used to promote any module to the top of the search chain. By using this command repeatedly, you can completely control the search chain.

Use the **.chain (List Debugger Extensions)** command to display a list of all loaded extension modules in their current search order.

If you attempt to execute an extension command that is not in any of the loaded extension modules, you will get an Export Not Found error message.

Writing New Debugger Extensions

12/5/2018 • 2 minutes to read • Edit Online

You can create your own debugging commands by writing an extension DLL. For example, you might want to write a command to display a complex data structure, or a command that will stop and start the target depending on the value of certain variables or memory locations.

There are two different types of debugger extensions:

- *DbgEng extensions*. These are based on the prototypes in the dbgeng.h header file, and also those in the wdbgexts.h header file.
- *WdbgExts extensions*. These are based on the prototypes in the wdbgexts.h header file alone.

For information about how to write debugger extensions, see Writing DbgEng Extensions and Writing WdbgExts Extensions.

Remote Debugging

12/5/2018 • 2 minutes to read • Edit Online

Remote user-mode debugging involves two computers: the *client* and the *server*. The server is the computer that runs the application to be debugged. The server also runs the user-mode debugger or a process server. The client is the computer that remotely controls the debugging session.

Remote kernel-mode debugging involves three computers: the *client*, the *server*, and the *target computer*. The target computer is the computer to be debugged. The server is a computer that runs the kernel debugger or a KD connection server. The client is the computer that remotely controls the debugging session.

Choosing the Best Remote Debugging Method

Remote Debugging Through the Debugger

Controlling the User-Mode Debugger from the Kernel Debugger

Remote Debugging Through Remote.exe

Process Servers (User Mode)

KD Connection Servers (Kernel Mode)

Repeaters

Advanced Remote Debugging Scenarios

Remote Debugging on Workgroup Computers

Choosing the Best Remote Debugging Method

1/3/2019 • 2 minutes to read • Edit Online

There are two primary methods of performing remote debugging, as well as several additional methods and a huge number of combination methods.

Here are some tips to help you choose the best technique.

- Remote debugging through the debugger is usually the best method. If you simply have one server and one client and they can freely connect to each other, the same debugger binaries are installed on both the client and the server, and the debugging technician who will be operating the client will be able to talk to someone in the room with the server, this is the recommended method.

  The client and the server can be running any version of Windows. They do not have to be running the same version as each other.

  If the client is unable to send a connection request to the server, but the server is able to send a request to the client, you can use remote debugging through the debugger with a *reverse connection* by using the **clicon** parameter.

- Remote debugging through remote.exe is used to remotely control a Command Prompt window. It can be used to remotely control KD, CDB, or NTSD. It cannot be used with WinDbg.

  If your client does not have copies of the debugger binaries, you must use the remote.exe method.

- **A process server** or a **KD connection server** can be used if the debugging technician will not be able to talk to someone in the room with the server. All the actual debugging work is done by the client (called the *smart client*); this removes the need to have a second person present at the server itself.

  Process servers are used for user-mode debugging; KD connection servers are used for kernel-mode debugging. Other than this distinction, they behave in similar ways.

  This method is also useful if the computer where the server will be running cannot handle heavy process loads, or if the technician running the client has access to symbol files or source files that are confidential and cannot be accessed by the server. However, this method is not as fast or efficient as remote debugging through the debugger. This method cannot be used for dump-file debugging.

  See Process Servers (User Mode) and KD Connection Servers (Kernel Mode) for details.

- **A repeater** is a lightweight proxy server that relays data between two computers. You can add a repeater between the client and the server if you are performing remote debugging through the debugger or if you are using a process server.

  Using a repeater may be necessary if your client and your server are unable to talk directly to each other, but can each access an outside computer. You can use reverse-connections with repeaters as well. It is even possible to use two repeaters in a row, but this is rarely necessary.

  See Repeaters for details.

- It is also possible to control CDB (or NTSD) from the kernel debugger. This is yet another form of remote debugging. See Controlling the User-Mode Debugger from the Kernel Debugger for details.

Variations on all of these methods are possible.

It is possible to chain several computers together to take advantage of more than one transport method. You can

create complicated transport sequences that take into account where the technician is, where the symbols are located, and whether there are firewalls preventing connections in certain directions. See Advanced Remote Debugging Scenarios for some examples.

You can even perform remote debugging on a single computer. For example, it might be useful to start a low-privilege process server and then connect to it with a high-privilege smart client.

Remote Debugging Through the Debugger

12/5/2018 • 2 minutes to read • Edit Online

Remote debugging directly through the debugger is usually the best and easiest method of performing remote debugging.

This technique involves running two debuggers at different locations. The debugger that is actually doing the debugging is called the *debugging server*. The debugger that is controlling the session from a distance is called the *debugging client*.

The two computers do not have to be running the same version of Windows; they can be running any version of Windows. The actual debuggers used need not be the same; a WinDbg debugging client can connect to a CDB debugging server, and so on.

However, it is best that the debugger binaries on the two computers both be from the same release of the Debugging Tools for Windows package, or at least both from recent releases.

To set up this remote session, the debugging server is set up first, and then the debugging client is activated. Any number of debugging clients can connect to a debugging server. A single debugger can turn itself into several debugging servers at the same time, to facilitate different kinds of connections.

However, no single debugger can be a debugging client and a debugging server simultaneously.

This section includes:

Activating a Debugging Server

Searching for Debugging Servers

Activating a Debugging Client

Client and Server Examples

Controlling a Remote Debugging Session

Activating a Debugging Server

1/3/2019 • 6 minutes to read • Edit Online

There are two ways to activate the debugging server. It can be activated when the debugger is started by using the **-server** command-line option in a elevated Command Prompt window (Run as Administrator). It can also be activated after the debugger is running. Start the debugger with elevated privileges (Run as Administrator), and enter the **.server** command.

Note You can activate a debugging server without having elevated privileges, and debugging clients will be able to connect to the server. However, clients will not be able to discover a debugging server unless it was activated with elevated privileges. For information about how to discover debugging servers, see Searching for Debugging Servers.

The debuggers support several transport protocols: named pipe (NPIPE), TCP, COM port, secure pipe (SPIPE), and secure sockets layer (SSL).

The general syntax for activating a debugging server depends on the protocol used.

```
Debugger -server npipe:pipe=PipeName[,hidden][,password=Password][,IcfEnable] [-noio] [Options]

Debugger -server tcp:port=Socket[,hidden][,password=Password][,ipversion=6][,IcfEnable] [-noio] [Options]

Debugger -server tcp:port=Socket,clicon=Client[,password=Password][,ipversion=6] [-noio] [Options]

Debugger -server com:port=COMPort,baud=BaudRate,channel=COMChannel[,hidden][,password=Password] [-noio]
[Options]

Debugger -server spipe:proto=Protocol,{certuser=Cert|machuser=Cert},pipe=PipeName[,hidden]
[,password=Password] [-noio] [Options]

Debugger -server ssl:proto=Protocol,{certuser=Cert|machuser=Cert},port=Socket[,hidden][,password=Password] [-
noio] [Options]

Debugger -server ssl:proto=Protocol,
{certuser=Cert|machuser=Cert},port=Socket,clicon=Client[,password=Password] [-noio] [Options]
```

Another method of activating a debugging server is to use the **.server (Create Debugging Server)** command after the debugger has already been started.

```
.server npipe:pipe=PipeName[,hidden][,password=Password][,IcfEnable]

.server tcp:port=Socket[,hidden][,password=Password][,ipversion=6][,IcfEnable]

.server tcp:port=Socket,clicon=Client[,password=Password][,ipversion=6]

.server com:port=COMPort,baud=BaudRate,channel=COMChannel[,hidden][,password=Password]

.server spipe:proto=Protocol,{certuser=Cert|machuser=Cert},pipe=PipeName[,hidden][,password=Password]

.server ssl:proto=Protocol,{certuser=Cert|machuser=Cert},port=Socket[,hidden][,password=Password]

.server ssl:proto=Protocol,{certuser=Cert|machuser=Cert},port=Socket,clicon=Client[,password=Password]
```

The parameters in the previous commands have the following possible values:

Debugger

Can be KD, CDB, NTSD, or WinDbg.

pipe= *PipeName*
When NPIPE or SPIPE protocol is used, *PipeName* is a string that will serve as the name of the pipe. Each pipe name should identify a unique debugging server. If you attempt to reuse a pipe name, you will receive an error message. *PipeName* must not contain spaces or quotation marks. *PipeName* can include a numerical **printf**-style format code, such as **%x** or **%d**. The debugger will replace this with the process ID of the debugger. A second such code will be replaced with the thread ID of the debugger.

Note You might need to enable file and printer sharing on the computer that is running the debugging server. In Control Panel, navigate to **Network and Internet > Network and Sharing Center> Advanced sharing settings**. Select **Turn on file and printer sharing**.

port= *Socket*
When TCP or SSL protocol is used, *Socket* is the socket port number.

It is also possible to specify a range of ports separated by a colon. The debugger will check each port in this range to see if it is free. If it finds a free port and no error occurs, the debugging server will be created. The debugging client will have to specify the actual port being used to connect to the server. To determine the actual port, use any of the methods described in **Searching for Debugging Servers**; when this debugging server is displayed, the port will be followed by two numbers separated by a colon. The first number will be the actual port used; the second can be ignored. For example, if the port was specified as **port=51:60**, and port 53 was actually used, the search results will show "port=53:60". (If you are using the **clicon** parameter to establish a reverse connection, the debugging client can specify a range of ports in this manner, while the server must specify the actual port used.)

clicon= *Client*
When TCP or SSL protocol is used and the **clicon** parameter is specified, a *reverse connection* will be opened. This means that the debugging server will try to connect to the debugging client, instead of letting the client initiate the contact. This can be useful if you have a firewall that is preventing a connection in the usual direction. *Client* specifies the network name or IP address of the computer on which the debugging client exists or will be created. The two initial backslashes (\) are optional.

Since the server is looking for one specific client, you cannot connect multiple clients to the server if you use this method. If the connection is refused or is broken you will have to restart the server connection. A reverse-connection server will not appear when another debugger displays all active servers.

Note When **clicon** is used, it is best to start the debugging client before the debugging server is created, although the usual order (server before client) is also permitted.

port= *COMPort*
When COM protocol is used, *COMPort* specifies the COM port to be used. The prefix "COM" is optional -- for example, both "com2" and "2" are acceptable.

baud= *BaudRate*
When COM protocol is used, *BaudRate* specifies the baud rate at which the connection will run. Any baud rate that is supported by the hardware is permitted.

channel= *COMChannel*
If COM protocol is used, *COMChannel* specifies the COM channel to be used in communicating with the debugging client. This can be any value between 0 and 254, inclusive. You can use a single COM port for multiple connections using different channel numbers. (This is different from the use of a COM ports for a debug cable -- in that situation you cannot use channels within a COM port.)

proto= *Protocol*
If SSL or SPIPE protocol is used, *Protocol* specifies the Secure Channel (S-Channel) protocol. This can be any one of the strings tls1, pct1, ssl2, or ssl3.

Cert
If SSL or SPIPE protocol is used, *Cert* specifies the certificate. This can either be the certificate name or the certificate's thumbprint (the string of hexadecimal digits given by the certificate's snapin). If the syntax **certuser=***Cert* is used, the debugger will look up the certificate in the system store (the default store). If the syntax **machuser=***Cert* is used, the debugger will look up the certificate in the machine store. The specified certificate must support server authentication.

hidden
Prevents the server from appearing when another debugger displays all active servers.

password= *Password*
Requires a client to supply the specified password in order to connect to the debugging session. *Password* can be any alphanumeric string, up to twelve characters in length.

Warning Using a password with TCP, NPIPE, or COM protocol only offers a small amount of protection, because the password is not encrypted. When a password is used with SSL or SPIPE protocol, it is encrypted. If you want to establish a secure remote session, you must use SSL or SPIPE protocol.

ipversion=6
(Debugging Tools for Windows 6.6.07 and earlier only) Forces the debugger to use IP version 6 rather than version 4 when using TCP to connect to the Internet. In Windows Vista and later versions, the debugger attempts to auto-default to IP version 6, making this option unnecessary.

-noio
If the debugging server is created with the **-noio** option, no input or output can be done through the server itself. The debugger will only accept input from the debugging client (plus any initial command or command script specified by the **-c** command-line option). All output will be directed to the debugging client. The **-noio** option is only available with KD, CDB, and NTSD. If NTSD is used for the server, no console window will be created at all.

IcfEnable
Causes the debugger to enable the necessary port connections for TCP or named pipe communication when the Internet Connection Firewall is active. By default, the Internet Connection Firewall disables the ports used by these protocols. When **IcfEnable** is used with a TCP connection, the debugger causes Windows to open the port specified by the *Socket* parameter. When **IcfEnable** is used with a named pipe connection, the debugger causes Windows to open the ports used for named pipes (ports 139 and 445). The debugger does not close these ports after the connection terminates.

Options
Any additional command-line parameters can be placed here. See Command-Line Options for a full list.

You can use the **.server** command to start multiple servers using different protocol options. This allows different kinds of debugging clients to join the session.

See Also

Controlling a Remote Debugging Session

.endsrv (End Debugging Server)

Searching for Debugging Servers

12/5/2018 • 2 minutes to read • Edit Online

You can use KD or CDB with the **-QR** command-line option to obtain a list of available debugging servers that are running on a network server.

This list may include servers that no longer exist but which were not shut down properly -- connecting to one of these generates an error message. The list will also include process servers and KD connection servers. The server type will be indicated in each case.

The syntax for this is as follows:

```
Debugger -QR \\Server
```

Debugger can be either KD or CDB -- the display will be the same in either case. The two backslashes (****) preceding *Server* are optional.

In WinDbg, you can use the **Connect to Remote Debugger Session** dialog box to browse a list of available servers. See File | Connect to Remote Session for details.

Note For a debugging server to be discoverable, it must be activated with elevated privileges. For more information, see Activating a Debugging Server.

Note If you are not logged on to the client computer with an account that has access to the server computer, you might need to provide a user name and password. On the client computer, in a Command Prompt window, enter the following command.

net use ***Server*\ipc$ /user:***UserName* where *Server* is the name of the server computer, and *UserName* is the name of an account that has access to the server computer.

When you are prompted, enter the password for *UserName*.

After this command succeeds, you can discover debugging servers (running on the server computer) by using **-QR** or **Connect to Remote Debugger Session**.

Activating a Debugging Client

12/5/2018 • 3 minutes to read • Edit Online

Once the debugging server has been activated, you can start a debugging client on another computer and connect to the debugging session.

There are two ways to start a debugging client: by using the -remote command-line option, or by using the WinDbg graphical interface.

The protocol of the client must match the protocol of the server. The general syntax for starting a debugging client depends on the protocol used. The following options exist:

```
Debugger -remote npipe:server=Server,pipe=PipeName[,password=Password]

Debugger -remote tcp:server=Server,port=Socket[,password=Password][,ipversion=6]

Debugger -remote tcp:clicon=Server,port=Socket[,password=Password][,ipversion=6]

Debugger -remote com:port=COMPort,baud=BaudRate,channel=COMChannel[,password=Password]

Debugger -remote spipe:proto=Protocol,
{certuser=Cert|machuser=Cert},server=Server,pipe=PipeName[,password=Password]

Debugger -remote ssl:proto=Protocol,
{certuser=Cert|machuser=Cert},server=Server,port=Socket[,password=Password]

Debugger -remote ssl:proto=Protocol,
{certuser=Cert|machuser=Cert},clicon=Server,port=Socket[,password=Password]
```

To use the graphical interface to connect to a remote debugging session, WinDbg must be in dormant mode -- it must either have been started with no command-line parameters, or it must have ended the previous debugging session. Select the **File | Connect to Remote Session** menu command, or press the CTRL+R shortcut key. When the **Connect to Remote Debugger Session** dialog box appears, enter one of the following strings into the **Connection string** text box:

```
npipe:server=Server,pipe=PipeName[,password=Password]

tcp:server=Server,port=Socket[,password=Password][,ipversion=6]

tcp:clicon=Server,port=Socket[,password=Password][,ipversion=6]

com:port=COMPort,baud=BaudRate,channel=COMChannel[,password=Password]

spipe:proto=Protocol,{certuser=Cert|machuser=Cert},server=Server,pipe=PipeName[,password=Password]

ssl:proto=Protocol,{certuser=Cert|machuser=Cert},server=Server,port=Socket[,password=Password]

ssl:proto=Protocol,{certuser=Cert|machuser=Cert},clicon=Server,port=Socket[,password=Password]
```

Alternatively, you can use the **Browse** button to locate active debugging servers. See File | Connect to Remote Session for details.

The parameters in the preceding commands have the following possible values:

Debugger

This does not have to be the same debugger as the one used by the debugging client -- WinDbg, KD, and CDB are all interchangeable for purposes of remote debugging through the debugger.

Server
This is the network name or IP address of the computer on which the debugging server was created. The two initial backslashes (\) are optional on the command line, but are not permitted in the WinDbg dialog box.

pipe= *PipeName*
If NPIPE or SPIPE protocol is used, *PipeName* is the name that was given to the pipe when the server was created.

If you are not logged on to the client computer with an account that has access to the server computer, you must provide a user name and password. On the client computer, in a Command Prompt window, enter the following command.

net use ***Server*\ipc$ /user:***UserName*

where *Server* is the name of the server computer, and *UserName* is the name of an account that has access to the server computer.

When you are prompted, enter the password for *UserName*.

After this command succeeds, you can activate a debugging client by using the **-remote** command-line option or by using the WinDbg graphical interface.

Note You might need to enable file and printer sharing on the server computer. In Control Panel, navigate to **Network and Internet > Network and Sharing Center> Advanced sharing settings**. Select **Turn on file and printer sharing**.

port= *Socket*
If TCP or SSL protocol is used, *Socket* is the same socket port number that was used when the server was created.

clicon
Specifies that the debugging server will try to connect to the client through a reverse connection. The client must use **clicon** if and only if the server is using **clicon**. In most cases, the debugging client is started before the debugging server when a reverse connection is used.

port= *COMPort*
If COM protocol is used, *COMPort* specifies the COM port to be used. The prefix "COM" is optional -- for example, both "com2" and "2" are acceptable.

baud= *BaudRate*
If COM protocol is used, *BaudRate* should match the baud rate chosen when the server was created.

channel= *COMChannel*
If COM protocol is used, *COMChannel* should match the channel number chosen when the server was created.

proto= *Protocol*
If SSL or SPIPE protocol is used, *Protocol* should match the secure protocol used when the server was created.

Cert
If SSL or SPIPE protocol is used, you should use the identical **certuser=***Cert* or **machuser=** *Cert* parameter that was used when the server was created.

password= *Password*
If a password was used when the server was created, *Password* must be supplied in order to create the debugging client. It must match the original password. Passwords are case-sensitive. If the wrong password is supplied, the error message will specify "Error 0x80004005." Passwords must be twelve characters or less in

length.

ipversion=6
(Debugging Tools for Windows 6.6.07 and earlier only) Forces the debugger to use IP version 6 rather than version 4 when using TCP to connect to the Internet. In Windows Vista and later versions, the debugger attempts to auto-default to IP version 6, making this option unnecessary.

Command-line options used to start new debugging sessions (like **-p**) cannot be used by the debugging client, but only by the server. Configuration options (like **-n**) will work from either the client or the server.

Client and Server Examples

12/5/2018 • 2 minutes to read • Edit Online

Suppose one person is running an application on a computer named *BOX17*. This application has problems, but the debugging technician is at a different site.

The first person sets up a debugging server using CDB on *BOX17*. The target application has a process ID of 122. TCP protocol is chosen, with a socket port number of 1025. The server is started by entering the following command in an elevated Command Prompt window (Run as Administrator):

```
E:\Debugging Tools for Windows> cdb -server tcp:port=1025 -p 122
```

On the other computer, the technician decides to use WinDbg as the debugging client. It can be started with this command:

```
G:\Debugging Tools> windbg -remote tcp:server=BOX17,port=1025
```

Here is another example. In this case, NPIPE protocol is chosen, and CDB is used instead of WinDbg. The first user chooses a pipe name. This can be any alphanumeric string -- in this example, "MainPipe". The first user opens an elevated Command Prompt window (Run as Administrator) and starts a debugging server by entering this command:

```
E:\Debugging Tools for Windows> cdb -server npipe:pipe=MainPipe -v winmine.exe
```

The technician is logged on to the client computer with an account that does not have access to the server computer. But the technician knows the username and password for an account that does have access to the server computer. The username for that account is Contoso. The technician enters the following command:

```
net use \\BOX17\ipc$ /user:Contoso
```

When prompted, the technician enters the password for the Contoso account.

The technician is not sure what name was used for the named pipe, so she queries BOX17 for available debugging servers.

```
G:\Debugging Tools> cdb -QR \\BOX17
Servers on \\BOX17:
Debugger Server - npipe:Pipe=MainPipe
Remote Process Server - npipe:Pipe=AnotherPipe
```

Two pipes are shown. However, only one is a debugging server -- the other is a process server, and we are not interested in that. So **MainPipe** must be the correct name. The technician uses the following command to start the debugging client:

```
G:\Debugging Tools> cdb -remote npipe:server=BOX17,pipe=MyPipe
```

Using a Secure Server

Here is an example of a secure server. This server uses secure sockets layer with an S-Channel protocol of TLS1. The debugger will look for the certificate in the machine store. The certificate is specified by its hexadecimal thumbprint.

```
D:\> cdb -server "ssl:proto=tls1,machuser=ab 38 f7 ae 13 20 ac da 05 14 65 60 30 83 7b 83 09 2c d2
34,port=1234" notepad.exe
```

Controlling a Remote Debugging Session

12/5/2018 • 3 minutes to read • Edit Online

Once the remote session has been started, commands can be entered into either the debugging server or the debugging client. If there are multiple clients, any of them can enter commands. Once ENTER is pressed, the command is transmitted to the debugging server and executed.

Whenever one user enters a command, all users will see the command itself and its output. If this command was issued from a debugging client, all other users will see an identification, preceding the command, of which user issued the command. Commands issued from the debugging server do not have this prefix.

After a command is executed by one user, other users who are connected through KD or CDB will not see a new command prompt. On the other hand, users of WinDbg will see the prompt in the bottom panel of the Debugger Command window continuously, even when the debugger engine is running. Neither of these should be a cause for alarm; any user can enter a command at any time, and the engine will execute these commands in the order they were received.

Actions made through the WinDbg interface will also be executed by the debugging server.

Communication Between Users

Whenever a new debugging client connects to the session, all other users will see a message that this client has connected. No message is displayed when a client disconnects.

The **.clients (List Debugging Clients)** command will list all clients currently connected to the debugging session.

The **.echo (Echo Comment)** command is useful for sending messages from one user to another.

WinDbg Workspaces

When WinDbg is being used as a debugging client, its workspace will only save values set through the graphical interface. Changes made through the Debugger Command window will not be saved. (This guarantees that only changes made by the local client will be reflected, since the Debugger Command window will accept input from all clients as well as the debugging server.)

File Paths

The symbol path, executable image path, and extension DLL path are all interpreted as file paths relative to the Debugging Tools for Windows installation folder on the debugging server.

When WinDbg is used as a debugging client, it has its own *local source path* as well. All source-related commands will access the source files on the local computer. Therefore, the proper paths have to be set on any client or server that will use source commands.

This multiple-path system allows a debugging client to use source-related commands without actually sharing the source files with other clients or with the server. This is useful if there are private or confidential source files which one of the users has access to.

Canceling the Debugging Server

The **.endsrv (End Debugging Server)** command can be used to terminate a debugging server. If the debugger has established multiple debugging servers, you can cancel some of them while leaving others running.

Terminating a server will prevent any future clients from attaching to it. It will not cut off any clients that are currently attached through the server.

Exiting the Debugger and Terminating the Session

To exit from one debugging client without terminating the server, you must issue a command from that specific client. If this client is KD or CDB, use the **CTRL+B** key to exit. If you are using a script to run KD or CDB, use **.remote_exit (Exit Debugging Client)**. If this client is WinDbg, choose **Exit** from the **File** menu to exit.

To terminate the entire session and exit the debugging server, use the **q (Quit)** command. This command can be entered from any server or client, and it will terminate the entire session for all users.

Controlling the User-Mode Debugger from the Kernel Debugger

12/5/2018 • 2 minutes to read • Edit Online

You can redirect the input and output from a user-mode debugger to a kernel debugger. This redirection enables the kernel debugger to control a specific user-mode debugging session that is occurring on the target computer.

You can use either KD or WinDbg as the kernel debugger. Note that many of the familiar features of WinDbg are not available in this scenario. For example, you cannot use the Locals window, the Disassembly window, or the Call Stack window, and you cannot step through source code. This is because WinDbg is only acting as a viewer for the debugger (NTSD or CDB) running on the target computer.

You can use either CDB or NTSD as the user-mode debugger. NTSD is the better choice, because it requires minimal resources from the processor and operating system of the computer whose application is being debugged. In fact, when NTSD is started under the control of the kernel debugger, no NTSD window is created. With NTSD, you can perform user-mode debugging through the serial port early in the boot phase and late into shutdown.

Note The **.shell** command is not supported when the output of a user-mode debugger is redirected to the kernel debugger.

This section includes the following:

- Starting the Debugging Session describes how to begin a session where the user-mode debugger is controlled from the kernel debugger.
- Switching Modes describes the four different modes that are involved, and how to alternate between them.
- When to Use This Technique describes scenarios where this technique is particularly useful.
- Combining This Method with Remote Debugging describes how to control the user-mode debugger from a kernel debugger, and use it as a debugging server at the same time. This combination can be useful if your user-mode symbols are located on a symbol server.

Starting the Debugging Session

12/5/2018 • 2 minutes to read • Edit Online

In this documentation of how to control user-mode debugging from the kernel debugger, *target application* refers to the user-mode application that is being debugged, *target computer* refers to the computer that contains the target application and the NTSD or CDB process, and *host computer* refers to the computer that contains the kernel debugger.

To begin using this technique, you must do the following. You can do steps 1 and 2 in either order.

1. Start NTSD or CDB on the target computer, with the -d command-line option.

   For example, you can attach to a running process by using the following syntax.

   ntsd -d [-y *UserSymbolPath***] -p** *PID*

   Or, you can start a new process as the target by using the following syntax.

   ntsd -d [-y *UserSymbolPath***]** *ApplicationName*

   If you are installing this as a postmortem debugger, you would use the following syntax.

   ntsd -d [-y *UserSymbolPath***]**

   For more information about this step, see Debugging a User-Mode Process Using CDB.

2. Start WinDbg or KD on the host computer, as if you were going to debug the target computer, but do not actually break in to the target computer. To use WinDbg, use the following syntax.

   windbg [-y *KernelSymbolPath***] [-k** *ConnectionOptions***]**

   For more information about this step, see Live Kernel-Mode Debugging Using WinDbg.

   Note If you use WinDbg as the kernel debugger, many of the familiar features of WinDbg are not available in this scenario. For example, you cannot use the Locals window, the Disassembly window, or the Call Stack window, and you cannot step through source code. This is because WinDbg is only acting as a viewer for the debugger (NTSD or CDB) running on the target computer.

3. If you have not set the user-mode symbol path, set it from the Input> prompt. If you have not set the kernel-mode symbol path, set it from the kd> prompt. For information on how to access these prompts and to switch between modes, see Switching Modes.

If you use CDB, the Command Prompt window that is associated with CDB remains locked and unavailable while debugging continues. If you use NTSD, no additional window is created, even though NTSD has a process ID associated with it on the target computer.

If you want to run the user-mode debugger from the kernel debugger while also using it as a debugging server, see Combining This Method with Remote Debugging.

Switching Modes

12/5/2018 • 3 minutes to read • Edit Online

When you control user-mode debugging from the kernel debugger, you encounter four different modes, and can switch between them in a variety of ways.

Note In describing this scenario, *target application* refers to the user-mode application that is being debugged, *target computer* refers to the computer that contains the target application and the CDB or NTSD process, and *host computer* refers to the computer that contains the kernel debugger.

The following four modes will be encountered:

User-mode debugging
The target computer and target application are frozen. The user-mode debugging prompt appears in the Debugger Command window of the kernel debugger. In WinDbg, the prompt in the lower panel of the WinDbg window displays Input>. You can enter commands at this prompt, as if they are entered during user-mode debugging, to analyze the target application's state or cause it to run or step through its execution. Symbol files, extension DLLs, and other files that the debugger accesses will be those files on the target computer, not the host computer.

Target application execution
The target computer is running, the target application is running, and the debugger is waiting. This mode is the same as letting the target run in ordinary debugging.

Sleep mode
The target computer is running, but the target application is frozen, and both debuggers are frozen. This mode is useful if you have to do something on the target computer but you do not want to change the state of the debugging session.

Kernel-mode debugging
The target computer and the target application are frozen. The kernel-mode debugging prompt kd> appears in the Debugger Command window of the kernel debugger. This mode is the typical kernel-mode debugging state.

The session begins in user-mode debugging mode. The following actions and events cause the mode to change:

- To switch from user-mode debugging to target application execution, use the **g (Go)** command at the `Input>` prompt.
- To temporarily switch from user-mode debugging to target application execution and then return to user-mode debugging, use a step, trace, or other temporary execution command. For a list of such commands, see Controlling the Target.
- To switch from user-mode debugging to sleep mode, use the **.sleep (Pause Debugger)** command. This command is timed. When the time expires, the system returns to user-mode debugging.
- To switch from user-mode debugging to kernel-mode debugging, use the **.breakin (Break to the Kernel Debugger)** command. Note that **.breakin** might fail with an access denied error if the calling process does not have administrator rights. In this case, switch to KD by issuing a short **.sleep** command and pressing CTRL+C.
- You can switch from target application execution to user-mode debugging only in certain environments. If the target computer is running Microsoft Windows XP or a later version of the Windows operating system, you can use the **!bpid** extension command. If you are using CDB (not NTSD), you can activate the CDB window on the target computer and press CTRL+C.

- If the target application hits a breakpoint, encounters an exception, encounters some other controlled event, or ends, the system switches from target application execution to user-mode debugging. You should plan such events in advance, especially when you are using NTSD. For more information about these events, see Using Breakpoints and Controlling Exceptions and Events.
- To switch from target application execution to kernel-mode debugging, press CTRL+C in the KD window, press CTRL+BREAK or click **Break** on the **Debug** menu in the WinDbg window, or press SYSRQ or ALT+SYSRQ on the target computer keyboard. (If your kernel debugger is KD and if you press CTRL+C at the same time that the kernel debugger is communicating with the user-mode debugger, the user-mode debugger might capture you pressing CTRL+C.)
- If the debugger encounters a kernel error or if you use the Breakin.exe tool, the system switches from target application execution to kernel-mode debugging.
- To switch from sleep mode to user-mode debugging, wait for the sleep time to expire, start a new CDB process on the target computer by using the -wake **command-line option**, or use the **.wake (Wake Debugger)** command in a different copy of CDB or NTSD on the target computer.
- To switch out of kernel-mode debugging, use the **g (Go)** command at the `kd>` prompt. This command returns to user-mode debugging or target application execution (whichever of the two was the most recently-used state).

When to Use This Technique

12/5/2018 • 2 minutes to read • Edit Online

There are several situations in which it is useful, or even necessary, to control user-mode debugging from the kernel debugger:

- When you need to perform user-mode debugging, but also need control over the Windows kernel that the user-mode target is running on or need to use some kernel-mode debugging features to analyze the problem.
- When your user-mode target is a Windows process such as CSRSS or WinLogon. For a detailed description of how to debug these targets, see Debugging CSRSS and Debugging WinLogon.
- When your user-mode target is a service application. For a detailed description of how to debug these targets, see Debugging a Service Application.

Combining This Method with Remote Debugging

12/5/2018 • 3 minutes to read • Edit Online

It is sometimes useful to control the user-mode debugger from the kernel debuggerand use the user-mode debugger as a debugging server at the same time.

For example, this configuration is useful when your user-mode symbols are located on a symbol server. In the standard configuration for controlling the user-mode debugger from a kernel debugger, the interaction of the two debuggers can lead to tiny lapses in synchronization, and these lapses can prevent symbol server authentication. The more complex configuration described here can avoid this problem.

Note In describing this scenario, *target application* refers to the user-mode application that is being debugged, *target computer* refers to the computer that contains the target application and the CDB or NTSD process, and *host computer* refers to the computer that contains the kernel debugger.

To use this technique, you must do the following:

1. Start NTSD or CDB on the target computer, with the -ddefer and -server command-line options, specifying the desired transport options. The -server option must be the first parameter on the command line.

   For example, you can attach to a running process by using the following syntax.

   ```
   ntsd -server ServerTransport -ddefer [-y UserSymbolPath] -p PID
   ```

   Or, you can start a new process as the target by using the following syntax.

   ```
   ntsd -server ServerTransport -ddefer [-y UserSymbolPath] ApplicationName
   ```

   If you are installing this as a postmortem debugger, you would use the following syntax. Note that you must manually edit the registry to install a postmortem debugger that includes the -server parameter; for details, see Enabling Postmortem Debugging.

   ```
   ntsd -server ServerTransport -ddefer [-y UserSymbolPath]
   ```

   For information about the available transport options, see **Activating a Debugging Server**.

2. Start WinDbg or KD on the host computer, as if you were going to debug the target computer, but do not actually break in to the target computer. To use WinDbg, use the following syntax.

   ```
   windbg [-y KernelSymbolPath] [-k ConnectionOptions]
   ```

   For more information about this step, see Live Kernel-Mode Debugging Using WinDbg

   .

3. Start WinDbg or CDB as a debugging client, with the same transport options used to start the server. This debugging client can be run on either the host computer or on a third computer.

   ```
   cdb -remote ClientTransport
   ```

For more information about this step, see **Activating a Debugging Client**.

4. Once the debuggers are running and the `Input>` prompt appears in the kernel debugger, use the **.sleep (Pause Debugger)** command to pause the debuggers and let the target computer run for a few seconds. This gives the target computer time to process the remote transport protocol, establishing the connection between the user-mode remote server and the remote client.

If you use CDB as the user-mode debugger, the Command Prompt window that is associated with CDB remains locked and unavailable while debugging continues. If you use NTSD, no additional window is created, even though NTSD has a process ID associated with it on the target computer.

The four modes and the methods of switching between them described in the topic Switching Modes apply in this combination scenario, with the following differences:

- There are two different user-mode debugging modes. When the target computer is running, the debugging server is controlled by the debugging client as in any other remote debugging session; this is called *remote-controlled user-mode debugging*. When the kernel-mode debugger is broken in to the target computer and the `Input>` prompt is showing, the user-mode debugger is controlled by the kernel debugger; this is called *kernel-controlled user-mode debugging*.
- These two modes are never available at the same time. When the kernel debugger is broken in to the target computer, even though the user-mode debugger may be active, the target computer is unable to process the remote transport protocol, and therefore the user-mode debugger will not be able to receive remote input across this connection.
- If your user-mode symbols are located on a symbol server, any debugger commands that require symbol access should be issued while in remote-controlled user-mode debugging mode.
- To switch from kernel-controlled user-mode debugging to remote-controlled user-mode debugging, use the **.sleep (Pause Debugger)** command. When the user-mode debugger wakes from the sleep command, it will be in remote-controlled user-mode debugging mode.
- To switch from remote-controlled user-mode debugging to kernel-mode debugging, enter any command from the `Input>` prompt. If this prompt is not visible, switch to kernel-mode debugging, and then use the **g (Go)** command at the `kd>` prompt.

Internally, a user-mode debugger started with -ddefer gives first priority to input from the debugging client, and second priority to input from the kernel debugger. However, there can never be a conflict between simultaneous inputs, because when the kernel debugger has broken in to the target computer, the remote connection is unavailable.

Remote Debugging Through Remote.exe

12/5/2018 • 2 minutes to read • Edit Online

Remote debugging through remote.exe involves running the debugger on the remote computer and running the remote.exe tool on the local computer.

The remote computer and the local computer can be running any Windows operating system.

Note Since remote.exe only works for console applications, it cannot be used to remotely control WinDbg.

This section includes:

The Remote.exe Utility

Starting a Remote.exe Session

Remote.exe Batch Files

The Remote.exe Utility

12/5/2018 • 2 minutes to read • Edit Online

The remote.exe utility is a versatile server/client tool that allows you to run command-line programs on remote computers.

Remote.exe provides remote network access by means of named pipes to applications that use STDIN and STDOUT for input and output. Users at other computers on a network, or connected by a direct-dial modem connection. can either view the remote session or enter commands themselves.

This utility has a large number of uses. For example, when you are developing software, you can compile code with the processor and resources of a remote computer while you perform other tasks on your computer. You can also use remote.exe to distribute the processing requirements for a particular task across several computers.

Please note that remote.exe does no security authorization, and will permit anyone running Remote.exe Client to connect to your Remote.exe Server. This leaves the account under which the Remote.exe Server was run open to anyone who connects.

Starting a Remote.exe Session

12/5/2018 • 2 minutes to read • Edit Online

There are two ways to start a remote.exe session with KD or CDB. Only the second of these methods works with NTSD.

Customizing Your Command Prompt Window

The Remote.exe Client and Remote.exe Server run in Command Prompt windows.

To prepare for the remote session, you should customize this window to increase its usability. Open a Command Prompt window. Right-click the title bar and select **Properties**. Select the **Layout** tab. Go to the section titled "Screen Buffer Size" and type **90** in the **Width** box and a value between **4000** and **9999** in the **Height** box. This enables scroll bars in the remote session on the kernel debugger.

Change the values for the height and width of the "Windows Size" section if you want to alter the shape of the command prompt. Select the **Options** tab. Enable the **Edit Options** quickedit mode and insert mode. This allows you to cut and paste information in the command prompt session. Click **OK** to apply the changes. Select the option to apply the changes to all future sessions when prompted.

Starting the Remote.exe Server: First Method

The general syntax for starting a Remote.exe Server is as follows:

```
remote /s "Command_Line" Unique_Id [/f Foreground_Color] [/b Background_Color]
```

This can be used to start KD or CDB on the remote computer, as in the following examples:

```
remote /s "KD [options]" MyBrokenBox

remote /s "CDB [options]" MyBrokenApp
```

This starts the Remote.exe Server in the Command Prompt window, and starts the debugger.

You cannot use this method to start NTSD directly, because the NTSD process runs in a different window than the one in which it was invoked.

Starting the Remote.exe Server: Second Method

There is an alternate method that can start a Remote.exe Server. This method involves first starting the debugger, and then using the **.remote (Create Remote.exe Server)** command to start the server.

Since the **.remote** command is issued after the debugger has started, this method works equally well with KD, CDB, and NTSD.

Here is an example. First, start the debugger in the normal fashion:

```
KD [options]
```

Once the debugger is running, use the **.remote** command:

```
.remote MyBrokenBox
```

This results in a KD process that is also a Remote.exe Server with an ID of "MyBrokenBox", exactly as in the first method.

One advantage of this method is that you do not have to decide in advance if you intend to use remote debugging. If you are debugging with one of the console debuggers and then decide that you would prefer someone in a remote location to take over, you can use the **.remote** command and then they can connect to your session.

Starting the Remote.exe Client

The general syntax for starting a Remote.exe Client is as follows:

```
remote /c ServerNetBIOSName Unique_ID [/l Lines_to_Get] [/f Foreground_Color] [/b Background_Color]
```dbgcmd

For example, if the "MyBrokenBox" session, described above, was started on a local host computer whose network name was "Server2", you can connect to it with the command:

```console
remote /c server2 MyBrokenBox
```

Anyone on the network with appropriate permission can connect to this debug session, as long as they know your machine name and the session ID.

Issuing Commands

Commands are issued through the Remote.exe Client and are sent to the Remote.exe Server. You can enter any command into the client as if you were directly entering it into the debugger.

To exit from the remote.exe session on the Remote.exe Client, enter the **@Q** command. This leaves the Remote.exe Server and the debugger running.

To end the server session, enter the **@K** command on the Remote.exe Server.

Remote.exe Batch Files

12/5/2018 • 2 minutes to read • Edit Online

As a more detailed example of remote debugging with remote.exe, assume the following about a local host computer in a three-computer kernel debugging scenario:

- Debugging needs to take place over a null-modem cable on COM2.
- The symbol files are in the folder c:\winnt\symbols.
- A log file called debug.log is created in **c:\temp**.

The log file holds a copy of everything you see on the Debug screen during your debug session. All input from the person doing the debugging, and all output from the kernel debugger on the target system, is written to that log file.

A sample batch file for running a debugging session on the local host is:

```
set _NT_DEBUG_PORT=com2
set _NT_DEBUG_BAUD_RATE=19200
set _NT_SYMBOL_PATH=c:\winnt\symbols
set _NT_LOG_FILE_OPEN=c:\temp\debug.log
remote /s "KD -v" debug
```

Note If this batch file is not in the same directory as Remote.exe, and Remote.exe is not in a directory listed in the system path, then you should give the full path to the utility when invoking Remote.exe in this batch file.

After this batch file is run, anyone with a Windows computer that is networked to the local host computer can connect to the debug session by using the following command:

```
remote /c computername debug
```

where *computername* is the NetBIOS name of the local host computer.

Process Servers (User Mode)

12/5/2018 • 2 minutes to read • Edit Online

Remote debugging through a process server involves running a small application called a *process server* on the server computer. Then a user-mode debugger is started on the client computer. Since this debugger will be doing all of the actual processing, it is called the *smart client*.

The Debugging Tools for Windows package includes a process server called DbgSrv (dbgsrv.exe) for use in user mode.

The two computers do not have to be running the same version of Windows; they can be running any version of Windows. However, the debugger binaries used on the client and the DbgSrv binary used on the server must be from the same release of the Debugging Tools for Windows package. This method cannot be used for dump-file debugging.

To set up this remote session, the process server is set up first, and then the smart client is activated. Any number of smart clients can operate through a single process server -- these debugging sessions will remain separate and will not interfere with each other. If a debugging session is ended, the process server will continue to run and can be used for new debugging sessions.

In this section

- **Activating a Process Server**
- **Searching for Process Servers**
- **Activating a Smart Client**
- Process Server Examples
- Controlling a Process Server Session

Activating a Process Server

3/1/2019 • 7 minutes to read • Edit Online

The process server that is included in Debugging Tools for Windows is called DbgSrv (dbgsrv.exe). To activate a process server, open an elevated Command Prompt window (Run as Administrator), and enter the **dbgsrv** command.

Note You can activate a process server without having elevated privileges, and debugging clients will be able to connect to the server. However, clients will not be able to discover a process server unless it was activated with elevated privileges. For information about how to discover debugging servers, see Searching for Process Servers.

DbgSrv supports several transport protocols: named pipe (NPIPE), TCP, COM port, secure pipe (SPIPE), and secure sockets layer (SSL).

```
dbgsrv -t npipe:pipe=PipeName[,hidden][,password=Password][,IcfEnable] [[-sifeo Executable] -c[s] AppCmdLine]
[-x | -pc]

dbgsrv -t tcp:port=Socket[,hidden][,password=Password][,ipversion=6][,IcfEnable] [[-sifeo Executable] -c[s]
AppCmdLine] [-x | -pc]

dbgsrv -t tcp:port=Socket,clicon=Client[,password=Password][,ipversion=6] [[-sifeo Executable] -c[s]
AppCmdLine] [-x | -pc]

dbgsrv -t com:port=COMPort,baud=BaudRate,channel=COMChannel[,hidden][,password=Password] [[-sifeo Executable]
-c[s] AppCmdLine] [-x | -pc]

dbgsrv -t spipe:proto=Protocol,{certuser=Cert|machuser=Cert},pipe=PipeName[,hidden][,password=Password] [[-
sifeo Executable] -c[s] AppCmdLine] [-x | -pc]

dbgsrv -t ssl:proto=Protocol,{certuser=Cert|machuser=Cert},port=Socket[,hidden][,password=Password] [[-sifeo
Executable] -c[s] AppCmdLine] [-x | -pc]

dbgsrv -t ssl:proto=Protocol,{certuser=Cert|machuser=Cert},port=Socket,clicon=Client[,password=Password] [[-
sifeo Executable] -c[s] AppCmdLine] [-x | -pc]
```

The parameters in the previous commands have the following possible values:

pipe= *PipeName*
When NPIPE or SPIPE protocol is used, *PipeName* is a string that will serve as the name of the pipe. Each pipe name should identify a unique process server. If you attempt to reuse a pipe name, you will receive an error message. *PipeName* must not contain spaces or quotation marks. *PipeName* can include a numerical **printf**-style format code, such as **%x** or **%d**. The process server will replace this with the process ID of DbgSrv. A second such code will be replaced with the thread ID of DbgSrv.

Note You might need to enable file and printer sharing on the computer that is running the process server. In Control Panel, navigate to **Network and Internet > Network and Sharing Center> Advanced sharing settings**. Select **Turn on file and printer sharing**.

port= *Socket*
When TCP or SSL protocol is used, *Socket* is the socket port number.

It is also possible to specify a range of ports separated by a colon. DbgSrv will check each port in this range to see if it is free. If it finds a free port and no error occurs, the process server will be created. The smart client will have to specify the actual port being used to connect to the server. To determine the actual port, use any of the methods described in **Searching for Process Servers**; when this process server is displayed, the port will be followed by

two numbers separated by a colon. The first number will be the actual port used; the second can be ignored. For example, if the port was specified as **port=51:60**, and port 53 was actually used, the search results will show "port=53:60". (If you are using the **clicon** parameter to establish a reverse connection, the smart client can specify a range of ports in this manner, while the process server must specify the actual port used.)

clicon= *Client*
When TCP or SSL protocol is used and the **clicon** parameter is specified, a *reverse connection* will be opened. This means that the process server will try to connect to the smart client, instead of letting the client initiate the contact. This can be useful if you have a firewall that is preventing a connection in the usual direction. *Client* specifies the network name or IP address of the computer on which the smart client exists or will be created. The two initial backslashes (\) are optional.

Since the process server is looking for one specific client, you cannot connect multiple clients to the server if you use this method. If the connection is refused or is broken you will have to restart the process server. A reverse-connection process server will not appear when someone uses the **-QR** command-line option to display all active servers.

Note When **clicon** is used, it is best to start the smart client before the process server is created, although the usual order (server before client) is also permitted.

port= *COMPort*
When COM protocol is used, *COMPort* specifies the COM port to be used. The prefix "COM" is optional -- for example, both "com2" and "2" are acceptable.

baud= *BaudRate*
When COM protocol is used, *BaudRate* specifies the baud rate at which the connection will run. Any baud rate that is supported by the hardware is permitted.

channel= *COMChannel*
If COM protocol is used, *COMChannel* specifies the COM channel to be used in communicating with the debugging client. This can be any value between 0 and 254, inclusive. You can use a single COM port for multiple connections using different channel numbers. (This is different from the use of a COM ports for a debug cable -- in that situation you cannot use channels within a COM port.)

proto= *Protocol*
If SSL or SPIPE protocol is used, *Protocol* specifies the Secure Channel (S-Channel) protocol. This can be any one of the strings tls1, pct1, ssl2, or ssl3.

Cert
If SSL or SPIPE protocol is used, *Cert* specifies the certificate. This can either be the certificate name or the certificate's thumbprint (the string of hexadecimal digits given by the certificate's snapin). If the syntax **certuser=***Cert* is used, the debugger will look up the certificate in the system store (the default store). If the syntax **machuser=***Cert* is used, the debugger will look up the certificate in the machine store. The specified certificate must support server authentication.

hidden
Prevents the process server from appearing when someone uses the **-QR** command-line option to display all active servers.

password= *Password*
Requires a smart client to supply the specified password in order to connect to the process server. *Password* can be any alphanumeric string, up to twelve characters in length.

Warning Using a password with TCP, NPIPE, or COM protocol only offers a small amount of protection, because the password is not encrypted. When a password is used with SSL or SPIPE protocol, it is encrypted. If you want to establish a secure remote session, you must use SSL or SPIPE protocol.

ipversion=6
(Debugging Tools for Windows 6.6.07 and earlier only) Forces the debugger to use IP version 6 rather than version 4 when using TCP to connect to the Internet. In Windows Vista and later versions, the debugger attempts to auto-default to IP version 6, making this option unnecessary.

IcfEnable
Causes the debugger to enable the necessary port connections for TCP or named pipe communication when the Internet Connection Firewall is active. By default, the Internet Connection Firewall disables the ports used by these protocols. When **IcfEnable** is used with a TCP connection, the debugger causes Windows to open the port specified by the *Socket* parameter. When **IcfEnable** is used with a named pipe connection, the debugger causes Windows to open the ports used for named pipes (ports 139 and 445). The debugger does not close these ports after the connection terminates.

-sifeo *Executable*
Suspends the Image File Execution Option (IFEO) value for the given image. *Executable* should include the file name of the executable image, including the file name extensions. The **-sifeo** option allows DbgSrv to be set as the IFEO debugger for an image created by the **-c** option, without causing recursive invocation due to the IFEO setting. This option can be used only if **-c** is used.

-c
Causes DbgSrv to create a new process. You can use this to create a process that you intend to debug. This is similar to spawning a new process from the debugger, except that this process will not be debugged when it is created. To debug this process, determine its PID and use the **-p** option when starting the smart client to debug this process.

s
Causes the newly-created process to be immediately suspended. If you are using this option, it is recommended that you use CDB as your smart client, and that you start the smart client with the **-pb** command-line option, in conjunction with **-p PID**. If you include the **-pb** option on the command line, the process will resume when the debugger attaches to it; otherwise you can resume the process with the **~*m** command.

AppCmdLine
Specifies the full command line of the process to be created. *AppCmdLine* can be either a Unicode or ASCII string, and can include any printable character. All text that appears after the **-c[s]** parameter will be taken to form the string *AppCmdLine*.

-x
Causes the remainder of the command line to be ignored. This option is useful if you are launching DbgSrv from an application that may append unwanted text to its command line.

-pc
Causes the remainder of the command line to be ignored. This option is useful if you are launching DbgSrv from an application that may append unwanted text to its command line. A syntax error results if **-pc** is the final element on the DbgSrv command line. Aside from this restriction, **-pc** is identical to **-x**.

You can start any number of process servers on one computer. However, this is generally unnecessary, since one process server can be used by any number of smart clients (each engaged in a different debugging session).

Searching for Process Servers

3/1/2019 • 2 minutes to read • Edit Online

You can use KD or CDB with the **-QR** command-line option to obtain a list of available process servers that are running on a network server computer.

This list may include servers that no longer exist but which were not shut down properly -- connecting to one of these generates an error message. The list will also include debugging servers and KD connection servers. The server type will be indicated in each case.

The syntax for this is as follows:

```
Debugger -QR \\Server
```

Debugger can be either KD or CDB -- the display will be the same in either case. The two backslashes (****) preceding *Server* are optional.

In WinDbg, you can use the **Connect to Remote Stub Server** dialog box to browse a list of available process servers. See File | Connect to Remote Stub for more details.

Note For a process server to be discoverable, it must be activated with elevated privileges. For more information, see Activating a Process Server.

Note If you are not logged on to the client computer with an account that has access to the server computer, you might need to provide a user name and password. On the client computer, in a Command Prompt window, enter the following command.

net use ***Server*\ipc$ /user:***UserName* where *Server* is the name of the server computer, and *UserName* is the name of an account that has access to the server computer.

When you are prompted, enter the password for *UserName*.

After this command succeeds, you can discover process servers (running on the server computer) by using **-QR** or **Connect to Remote Stub**.

Activating a Smart Client

3/1/2019 • 4 minutes to read • Edit Online

Once the DbgSrv process server has been activated, you can create a smart client on another computer and begin a debugging session.

There are two ways to start a smart client: by starting CDB or WinDbg with the -premote command-line option, or by using the WinDbg graphical interface.

The protocol of the smart client must match the protocol of the process server. The general syntax for starting a smart client depends on the protocol used. The following options exist:

```
Debugger -premote npipe:server=Server,pipe=PipeName[,password=Password] [Options]

Debugger -premote tcp:server=Server,port=Socket[,password=Password][,ipversion=6] [Options]

Debugger -premote tcp:clicon=Server,port=Socket[,password=Password][,ipversion=6] [Options]

Debugger -premote com:port=COMPort,baud=BaudRate,channel=COMChannel[,password=Password] [Options]

Debugger -premote spipe:proto=Protocol,
{certuser=Cert|machuser=Cert},server=Server,pipe=PipeName[,password=Password] [Options]

Debugger -premote ssl:proto=Protocol,
{certuser=Cert|machuser=Cert},server=Server,port=Socket[,password=Password] [Options]

Debugger -premote ssl:proto=Protocol,
{certuser=Cert|machuser=Cert},clicon=Server,port=Socket[,password=Password] [Options]
```

To use the graphical interface to connect to a process server, WinDbg must be in dormant mode -- it must either have been started with no command-line parameters, or it must have ended the previous debugging session. Select the **File | Connect to Remote Stub** menu command. When the **Connect to Remote Stub Server** dialog box appears, enter one of the following strings into the **Connection string** text box:

```
npipe:server=Server,pipe=PipeName[,password=Password]

tcp:server=Server,port=Socket[,password=Password][,ipversion=6]

tcp:clicon=Server,port=Socket[,password=Password][,ipversion=6]

com:port=COMPort,baud=BaudRate,channel=COMChannel[,password=Password]

spipe:proto=Protocol,{certuser=Cert|machuser=Cert},server=Server,pipe=PipeName[,password=Password]

ssl:proto=Protocol,{certuser=Cert|machuser=Cert},server=Server,port=Socket[,password=Password]

ssl:proto=Protocol,{certuser=Cert|machuser=Cert},clicon=Server,port=Socket[,password=Password]
```

Alternatively, you can use the **Browse** button to locate active process servers. See File | Connect to Remote Stub for details.

The parameters in the preceding commands have the following possible values:

Debugger
This can be CDB or WinDbg.

Server
This is the network name or IP address of the computer on which the process server was created. The two initial backslashes (\) are optional on the command line, but are not permitted in the WinDbg dialog box.

pipe= *PipeName*
If NPIPE or SPIPE protocol is used, *PipeName* is the name that was given to the pipe when the process server was created.

If you are not logged on to the client computer with an account that has access to the server computer, you must provide a user name and password. On the client computer, in a Command Prompt window, enter the following command.

net use ***Server*\ipc$ /user:***UserName*

where *Server* is the name of the server computer, and *UserName* is the name of an account that has access to the server computer.

When you are prompted, enter the password for *UserName*.

After this command succeeds, you can activate a smart client by using the **-premote** command-line option or by using the WinDbg graphical interface.

Note You might need to enable file and printer sharing on the server computer. In Control Panel, navigate to **Network and Internet > Network and Sharing Center> Advanced sharing settings**. Select **Turn on file and printer sharing**.

port= *Socket*
If TCP or SSL protocol is used, *Socket* is the same socket port number that was used when the process server was created.

clicon
Specifies that the process server will try to connect to the smart client through a reverse connection. The client must use **clicon** if and only if the server is using **clicon**. In most cases, the smart client is started before the process server when a reverse connection is used.

port= *COMPort*
If COM protocol is used, *COMPort* specifies the COM port to be used. The prefix "COM" is optional -- for example, both "com2" and "2" are acceptable.

baud= *BaudRate*
If COM protocol is used, *BaudRate* should match the baud rate chosen when the process server was created.

channel= *COMChannel*
If COM protocol is used, *COMChannel* should match the channel number chosen when the process server was created.

proto= *Protocol*
If SSL or SPIPE protocol is used, *Protocol* should match the secure protocol used when the process server was created.

Cert
If SSL or SPIPE protocol is used, you should use the identical **certuser=***Cert* or **machuser=***Cert* parameter that was used when the process server was created.

password= *Password*
If a password was used when the process server was created, *Password* must be supplied in order to create the smart client. It must match the original password. Passwords are case-sensitive. If the wrong password is supplied, the error message will specify "Error 0x80004005."

ipversion=6
(Debugging Tools for Windows 6.6.07 and earlier only) Forces the debugger to use IP version 6 rather than version 4 when using TCP to connect to the Internet. In Windows Vista and later versions, the debugger attempts to auto-default to IP version 6, making this option unnecessary.

Options
Any additional command-line parameters can be placed here. See Command-Line Options for a full list. If you are using CDB, this must specify the process you wish to debug. If you are using WinDbg, you can specify the process on the command line or through the graphical interface.

Since the process server simply acts as a gateway for the smart client, the additional *Options* will be the same as those you would use if you were starting a user-mode debugger on the same machine as the target application.

If you are using the **-premote** option with **.attach (Attach to Process)** or **.create (Create Process)**, the parameters are the same as those listed above.

Troubleshooting

If you see this message: *The client is not using same version of remoting protocol as the server* this indicates that the version of DbgSrv that you are attempting to connect to is using a different protocol version than the version of WinDbg.

It is uncommon that protocol changes are made. When this does happen, be sure you are using the matching versions of the latest available version of DbgSrv and WinDbg or WinDbg Preview. For information on downloading the latest version, see Download Debugging Tools for Windows.

Process Server Examples

12/5/2018 • 2 minutes to read • Edit Online

Suppose one person is running an application on a computer named \\BOX17. This application has problems, but the debugging technician is at a different site.

The first person sets up a process server using DbgSrv on \\BOX17. The target application has a process ID of 122. TCP protocol is chosen, with a socket port number of 1025. The server is started with the following command:

```
E:\Debugging Tools for Windows> dbgsrv -t tcp:port=1025
```

On the other computer, the technician starts WinDbg as a smart client with this command:

```
G:\Debugging Tools> windbg -premote tcp:server=BOX17,port=1025 -p 122
```

Here is another example. In this case, NPIPE protocol is chosen, and CDB is used instead of WinDbg. The first user chooses a pipe name. This can be any alphanumeric string -- in this example, "AnotherPipe". The first user opens an elevated Command Prompt window (Run as Administrator) and starts a debugging server by entering this command:

```
E:\Debugging Tools for Windows> dbgsrv -t npipe:pipe=AnotherPipe
```

The technician is logged on to the client computer with an account that does not have access to the server computer. But the technician knows the username and password for an account that does have access to the server computer. The username for that account is Contoso. The technician enters the following command:

```
net use \\BOX17\ipc$ /user:Contoso
```

When prompted, the technician enters the password for the Contoso account.

The technician is not sure what name was used for the named pipe, so she queries BOX17 for process servers:

```
G:\Debugging Tools> cdb -QR \\BOX17
Servers on \\BOX17:
Debugger Server - npipe:Pipe=MainPipe
Remote Process Server - npipe:Pipe=AnotherPipe
```

Two pipes are shown. However, only one is a process server -- the other is a debugging server, and we are not interested in that. So **AnotherPipe** must be the correct name. The technician enters the following command to start the smart client:

```
G:\Debugging Tools> cdb -premote npipe:server=BOX17,pipe=AnotherPipe -v sol.exe
```

For a more complicated example using a process server, see Symbols in the Middle.

Controlling a Process Server Session

12/5/2018 • 2 minutes to read • Edit Online

Once the remote session has been started, the smart client can be used as if it were debugging a target application on a single machine. All commands will behave as they would in this situation, except that paths are relative to the smart client's computer.

Using WinDbg as a Smart Client

After WinDbg is started as a smart client for a user-mode process server, it will remain attached to the process server permanently. If the debugging session is ended, the File | Attach to a Process menu command or the **.tlist (List Process IDs)** command will display all processes running on the computer running the process server. WinDbg can attach to any of these processes.

The File | Open Executable command cannot be used. A new process can only be spawned if it is included on the WinDbg command line.

In this situation, WinDbg will not be able to debug processes on the computer where it is running, nor will it be able to start a kernel debugging session.

Ending the Session

CDB or WinDbg can exit or end the debugging session in the normal fashion. See Ending a Debugging Session in WinDbg for details. The process server will remain in operation and can be re-used as many times as desired. (It can also be used by for any number of simultaneous debugging sessions.)

The process server can be terminated from either computer. To terminate it from the smart client, use the **.endpsrv (End Process Server)** command. To terminate the process server from the computer it is running on, use Task Manager to end the dbgsrv.exe process.

KD Connection Servers (Kernel Mode)

12/5/2018 • 2 minutes to read • Edit Online

Kernel-mode remote debugging through a KD connection server involves running a small application called a *KD connection server* on the server. Then a kernel-mode debugger is started on the client. Since this debugger will be doing all of the actual processing, it is called the *smart client*.

The Debugging Tools for Windows package includes a KD connection server called KdSrv (kdsrv.exe).

The two computers do not have to be running the same version of Windows; they can be running any version of Windows. However, the debugger binaries used on the client and the KdSrv binary used on the server must be from the same release of the Debugging Tools for Windows package. This method cannot be used for dump-file debugging.

To set up this remote session, the KD connection server is set up first, and then the smart client is activated. Any number of smart clients can operate through a single KD connection server, but they must each be connected to a different kernel debugging session.

This section includes:

Activating a KD Connection Server

Searching for KD Connection Servers

Activating a Smart Client (Kernel Mode)

KD Connection Server Examples

Controlling a KD Connection Server Session

Activating a KD Connection Server

1/3/2019 • 5 minutes to read • Edit Online

The KD connection server that is included in Debugging Tools for Windows is called KdSrv (kdsrv.exe). To activate a KD connection server, open an elevated Command Prompt window (Run as Adminstrator), and enter the **kdsrv** command.

Note You can activate a KD connection server without having elevated privileges, and debugging clients will be able to connect to the server. However, clients will not be able to discover a KD connection server unless it was activated with elevated privileges. For information about how to discover debugging servers, see **Searching for KD Connection Servers**.

KdSrv supports several transport protocols: named pipe (NPIPE), TCP, COM port, secure pipe (SPIPE), and secure sockets layer (SSL).

The syntax for the KdSrv command line depends on the protocol used. The following options exist:

```
kdsrv -t npipe:pipe=PipeName[,hidden][,password=Password][,IcfEnable]

kdsrv -t tcp:port=Socket[,hidden][,password=Password][,ipversion=6][,IcfEnable]

kdsrv -t tcp:port=Socket,clicon=Client[,password=Password][,ipversion=6]

kdsrv -t com:port=COMPort,baud=BaudRate,channel=COMChannel[,hidden][,password=Password]

kdsrv -t spipe:proto=Protocol,{certuser=Cert|machuser=Cert},pipe=PipeName[,hidden][,password=Password]

kdsrv -t ssl:proto=Protocol,{certuser=Cert|machuser=Cert},port=Socket[,hidden][,password=Password]

kdsrv -t ssl:proto=Protocol,{certuser=Cert|machuser=Cert},port=Socket,clicon=Client[,password=Password]
```

The parameters in the previous commands have the following possible values:

pipe= *PipeName*
When NPIPE or SPIPE protocol is used, *PipeName* is a string that will serve as the name of the pipe. Each pipe name should identify a unique process server. If you attempt to reuse a pipe name, you will receive an error message. *PipeName* must not contain spaces or quotation marks. *PipeName* can include a numerical **printf**-style format code, such as **%x** or **%d**. This will be replaced by the process ID of KdSrv. A second such code will be replaced by the thread ID of KdSrv.

port= *Socket*
When TCP or SSL protocol is used, *Socket* is the socket port number.

It is also possible to specify a range of ports separated by a colon. KdSrv will check each port in this range to see if it is free. If it finds a free port and no error occurs, the KD connection server will be created. The smart client will have to specify the actual port being used to connect to the server. To determine the actual port, use any of the methods described in **Searching for KD Connection Servers**; when this KD connection server is displayed, the port will be followed by two numbers separated by a colon. The first number will be the actual port used; the second can be ignored. For example, if the port was specified as **port=51:60**, and port 53 was actually used, the search results will show "port=53:60". (If you are using the **clicon** parameter to establish a reverse connection, the smart client can specify a range of ports in this manner, while the KD connection server must specify the actual port used.)

clicon= *Client*
When TCP or SSL protocol is used and the **clicon** parameter is specified, a *reverse connection* will be opened. This means that the KD connection server will try to connect to the smart client, instead of letting the client initiate the contact. This can be useful if you have a firewall that is preventing a connection in the usual direction. *Client* specifies the network name or IP address of the computer on which the smart client exists or will be created. The two initial backslashes (\) are optional.

Since the KD connection server is looking for one specific client, you cannot connect multiple clients to the server if you use this method. If the connection is refused or is broken you will have to restart the process server. A reverse-connection KD connection server will not appear when someone uses the **-QR** command-line option to display all active servers.

Note When **clicon** is used, it is best to start the smart client before the KD connection server is created, although the usual order (server before client) is also permitted.

port= *COMPort*
When COM protocol is used, *COMPort* specifies the COM port to be used. The prefix "COM" is optional -- for example, both "com2" and "2" are acceptable.

baud= *BaudRate*
When COM protocol is used, *BaudRate* specifies the baud rate at which the connection will run. Any baud rate that is supported by the hardware is permitted.

channel= *COMChannel*
If COM protocol is used, *COMChannel* specifies the COM channel to be used in communicating with the debugging client. This can be any value between 0 and 254, inclusive. You can use a single COM port for multiple connections using different channel numbers. (This is different from the use of a COM ports for a debug cable -- in that situation you cannot use channels within a COM port.)

proto= *Protocol*
If SSL or SPIPE protocol is used, *Protocol* specifies the Secure Channel (S-Channel) protocol. This can be any one of the strings tls1, pct1, ssl2, or ssl3.

Cert
If SSL or SPIPE protocol is used, *Cert* specifies the certificate. This can either be the certificate name or the certificate's thumbprint (the string of hexadecimal digits given by the certificate's snapin). If the syntax **certuser=***Cert* is used, the debugger will look up the certificate in the system store (the default store). If the syntax **machuser=***Cert* is used, the debugger will look up the certificate in the machine store. The specified certificate must support server authentication.

hidden
Prevents the KD connection server from appearing when someone uses the **-QR** command-line option to display all active servers.

password= *Password*
Requires a smart client to supply the specified password in order to connect to the KD connection server. *Password* can be any alphanumeric string, up to twelve characters in length.

Warning Using a password with TCP, NPIPE, or COM protocol only offers a small amount of protection, because the password is not encrypted. When a password is used with SSL or SPIPE protocol, it is encrypted. If you want to establish a secure remote session, you must use SSL or SPIPE protocol.

ipversion=6
(Debugging Tools for Windows 6.6.07 and earlier only) Forces the debugger to use IP version 6 rather than version 4 when using TCP to connect to the Internet. In Windows Vista and later versions, the debugger attempts to auto-default to IP version 6, making this option unnecessary.

IcfEnable

Causes the debugger to enable the necessary port connections for TCP or named pipe communication when the Internet Connection Firewall is active. By default, the Internet Connection Firewall disables the ports used by these protocols. When **IcfEnable** is used with a TCP connection, the debugger causes Windows to open the port specified by the *Socket* parameter. When **IcfEnable** is used with a named pipe connection, the debugger causes Windows to open the ports used for named pipes (ports 139 and 445). The debugger does not close these ports after the connection terminates.

Searching for KD Connection Servers

12/5/2018 • 2 minutes to read • Edit Online

You can use KD or CDB with the **-QR** command-line option to obtain a list of available KD connection servers that are running on a network server.

This list may include servers that no longer exist but which were not shut down properly -- connecting to one of these generates an error message. The list will also include debugging servers and user-mode process servers. The server type will be indicated in each case.

The syntax for this is as follows:

```
Debugger -QR \\Server
```

Debugger can be either KD or CDB -- the display will be the same in either case. The two backslashes (****) preceding *Server* are optional.

In WinDbg, you can use the **Connect to Remote Stub Server** dialog box to browse a list of available KD connection servers. See File | Connect to Remote Stub for more details.

Note For a KD connection server to be discoverable, it must be activated with elevated privileges. For more information, see Activating a KD Connection Server.

Note If you are not logged on to the client computer with an account that has access to the server computer, you might need to provide a user name and password. On the client computer, in a Command Prompt window, enter the following command.

net use ***Server*\ipc$ /user:***UserName* where *Server* is the name of the server computer, and *UserName* is the name of an account that has access to the server computer.

When you are prompted, enter the password for *UserName*.

After this command succeeds, you can discover KD connection servers (running on the server computer) by using **-QR** or **Connect to Remote Stub**.

Activating a Smart Client (Kernel Mode)

12/5/2018 • 4 minutes to read • Edit Online

Once the KD connection server has been activated, you can create a smart client on another computer and begin a debugging session.

There are two ways to start a smart client: by starting KD or WinDbg with the kernel protocol **kdsrv**, or by using the WinDbg graphical interface.

You need to specify the remote transfer protocol used by the KD connection server. You can also specify the protocol for the actual kernel connection between the KD connection server and the target computer, or you can use the default.

The general syntax for starting a smart client depends on the protocol used. The following options exist:

```
Debugger -k kdsrv:server=@{npipe:server=Server,pipe=PipeName[,password=Password]},trans=@{ConnectType}
[Options]

Debugger -k kdsrv:server=@{tcp:server=Server,port=Socket[,password=Password]
[,ipversion=6]},trans=@{ConnectType} [Options]

Debugger -k kdsrv:server=@{tcp:clicon=Server,port=Socket[,password=Password]
[,ipversion=6]},trans=@{ConnectType} [Options]

Debugger -k
kdsrv:server=@{com:port=COMPort,baud=BaudRate,channel=COMChannel[,password=Password]},trans=@{ConnectType}
[Options]

Debugger -k kdsrv:server=@{spipe:proto=Protocol,
{certuser=Cert|machuser=Cert},server=Server,pipe=PipeName[,password=Password]},trans=@{ConnectType} [Options]

Debugger -k kdsrv:server=@{ssl:proto=Protocol,
{certuser=Cert|machuser=Cert},server=Server,port=Socket[,password=Password]},trans=@{ConnectType} [Options]

Debugger -k kdsrv:server=@{ssl:proto=Protocol,
{certuser=Cert|machuser=Cert},clicon=Server,port=Socket[,password=Password]},trans=@{ConnectType} [Options]
```

To use the graphical interface to connect to a KD connection server, WinDbg must be in dormant mode -- it must either have been started with no command-line parameters, or it must have ended the previous debugging session. Select the **File | Connect to Remote Stub** menu command. When the **Connect to Remote Stub Server** dialog box appears, enter one of the following strings into the **Connection string** text box:

```
npipe:server=Server,pipe=PipeName[,password=Password]

tcp:server=Server,port=Socket[,password=Password][,ipversion=6]

tcp:clicon=Server,port=Socket[,password=Password][,ipversion=6]

com:port=COMPort,baud=BaudRate,channel=COMChannel[,password=Password]

spipe:proto=Protocol,{certuser=Cert|machuser=Cert},server=Server,pipe=PipeName[,password=Password]

ssl:proto=Protocol,{certuser=Cert|machuser=Cert},server=Server,port=Socket[,password=Password]

ssl:proto=Protocol,{certuser=Cert|machuser=Cert},clicon=Server,port=Socket[,password=Password]
```

Alternatively, you can use the **Browse** button to locate active KD connection servers. See File | Connect to Remote

Stub for details.

The parameters in the preceding commands have the following possible values:

Debugger
This can be KD or WinDbg.

Server
This is the network name or IP address of the computer on which the KD connection server was created.The two initial backslashes (\) are optional on the command line, but are not permitted in the WinDbg dialog box.

pipe= *PipeName*
If NPIPE or SPIPE protocol is used, *PipeName* is the name that was given to the pipe when the KD connection server was created.

If you are not logged on to the client computer with an account that has access to the server computer, you must provide a user name and password. On the client computer, in a Command Prompt window, enter the following command.

net use ***Server*\ipc$ /user:***UserName*

where *Server* is the name of the server computer, and *UserName* is the name of an account that has access to the server computer.

When you are prompted, enter the password for *UserName*.

After this command succeeds, you can activate a smart client by using **-k kdsrv** or by using the WinDbg graphical interface.

port= *Socket*
If TCP or SSL protocol is used, *Socket* is the same socket port number that was used when the KD connection server was created.

clicon
Specifies that the KD connection server will try to connect to the smart client through a reverse connection. The client must use **clicon** if and only if the server is using **clicon**. In most cases, the smart client is started before the KD connection server when a reverse connection is used.

port= *COMPort*
If COM protocol is used, *COMPort* specifies the COM port to be used. The prefix "COM" is optional -- for example, both "com2" and "2" are acceptable.

baud= *BaudRate*
If COM protocol is used, *BaudRate* should match the baud rate chosen when the KD connection server was created.

channel= *COMChannel*
If COM protocol is used, *COMChannel* should match the channel number chosen when the KD connection server was created.

proto= *Protocol*
If SSL or SPIPE protocol is used, *Protocol* should match the secure protocol used when the KD connection server was created.

Cert
If SSL or SPIPE protocol is used, you should use the identical **certuser=***Cert* or **machuser=***Cert* parameter that was used when the KD connection server was created.

password= *Password*

If a password was used when the KD connection server was created, *Password* must be supplied in order to create the smart client. It must match the original password. Passwords are case-sensitive. If the wrong password is supplied, the error message will specify "Error 0x80004005."

ipversion=6
(Debugging Tools for Windows 6.6.07 and earlier only) Forces the debugger to use IP version 6 rather than version 4 when using TCP to connect to the Internet. In Windows Vista and later versions, the debugger attempts to auto-default to IP version 6, making this option unnecessary.

trans=@{ *ConnectType* **}**
Tells the debugger how to connect to the target. The following kernel connection protocols are permitted:

```
com:port=ComPort,baud=BaudRate
1394:channel=1394Channel[,symlink=1394Protocol]
usb2:targetname=String
com:pipe,port=\\VMHost\pipe\PipeName[,resets=0][,reconnect]
com:modem
```

For information about these protocols, see Getting Set Up for Debugging. You can omit any of the parameters for these protocols -- for example, you can say **trans=@{com:}** -- and the debugger will default to the values specified by the environment variables on the computer where KdSrv is running.

Options
Any additional command-line parameters can be placed here. See Command-Line Options for a full list.

Since the KD connection server simply acts as a gateway for the smart client, the additional *Options* will be the same as those you would use if you were starting a kernel debugger on computer where KdSrv is running. The exception to this is any option that specifies a path or filename will be taken as a path on the computer where the smart client is running.

KD Connection Server Examples

12/5/2018 • 2 minutes to read • Edit Online

Suppose a debugging technician is not present at the site where the computer to be debugged is located. The debugging technician asks someone at this site to connect this target computer to some other computer with a debug cable.

Let this other computer be at IP address 127.0.0.42. The debug cable connects COM1 on this computer to whichever port has been debug-enabled on the target computer. The KD connection server is started with this command:

```
E:\Debugging Tools for Windows> kdsrv -t tcp:port=1027
```

Then at the other location, the technician starts WinDbg as a smart client with this command:

```
G:\Debugging Tools> windbg -k
kdsrv:server=@{tcp:server=127.0.0.42,port=1027},trans=@{com:port=com1,baud=57600} -y SymbolPath
```

The symbol path will be relative to the computer where the smart client is running.

Here is another example. In this case, NPIPE protocol is chosen, and KD is used instead of WinDbg. The first user chooses a pipe name. This can be any alphanumeric string -- in this example, "KernelPipe". The first user opens an elevated Command Prompt window (Run as Administrator) and starts a debugging server by entering these commands:

```
E:\Debugging Tools for Windows> set _NT_DEBUG_PORT=com1
E:\Debugging Tools for Windows> kdsrv -t npipe:pipe=KernelPipe
```

The technician is logged on to the client computer with an account that does not have access to the server computer. But the technician knows the username and password for an account that does have access to the server computer. The username for that account is Contoso. The technician enters the following command:

```
net use \\BOX17\ipc$ /user:Contoso
```

When prompted, the technician enters the password for the Contoso account.

The technician is not sure what name was used for the named pipe, so she queries 127.0.0.42 for KD connection servers:

```
G:\Debugging Tools> cdb -QR 127.0.0.42
Servers on 127.0.0.42:
Debugger Server - npipe:Pipe=MainPipe
Remote Process Server - npipe:Pipe=AnotherPipe
Remote Kernel Debugger Server - npipe:Pipe=KernelPipe
```

Three pipes are shown. However, only one is a KD connection server -- the others are a debugging server and a user-mode process server. The technician enters the following command can to start the smart client:

```
G:\Debugging Tools> kd -k kdsrv:server=@{npipe:server=127.0.0.42,pipe=KernelPipe},trans=@{com:baud=57600} -y
SymbolPath
```

Notice that although the baud rate is specified, the port is not. This causes the debugger to default to the port specified by _NT_DEBUG_PORT on the computer where KdSrv is running.

Controlling a KD Connection Server Session

12/5/2018 • 2 minutes to read • Edit Online

Once the remote session has been started, the smart client can be used as if it were debugging the target computer from the computer where the KD connection server is running. All commands will behave as they would in this situation, except that paths are relative to the smart client's computer.

Using WinDbg as a Smart Client

After WinDbg is started as a smart client for a KD connection server, you can use the Debug | Stop Debugging command to end the debugging session. At that point, WinDbg will enter dormant mode and will no longer be connected to the KD connection server. All subsequent debugging will be done on the computer where WinDbg is running. You cannot reattach to the KD connection serve by using File | Kernel Debug -- this can only be done from the command line.

Ending the Session

KD or WinDbg can exit or end the debugging session in the normal fashion. See Ending a Debugging Session in WinDbg

for details. The KD connection server will remain in operation and can be re-used as many times as desired. (It can also be used by for any number of simultaneous debugging sessions.)

The KD connection server can be terminated from either computer. To terminate it from the smart client, use the **.endpsrv (End Process Server)** command. To terminate the KD connection server from the computer it is running on, use Task Manager to end the kdsrv.exe process.

Repeaters

12/5/2018 • 2 minutes to read • Edit Online

A *repeater* is a lightweight proxy server that runs on a computer and relays data between two other computers. The repeater does not process the data in any way. The two other computers barely notice the repeater; from their perspective it seems as if they are directly connected to each other.

The processes running on these two other computers are called the *server* and the *client*. There is not any fundamental difference between them from the repeater's point of view, except that in most cases the server is started first, then the repeater, and finally the client.

The Debugging Tools for Windows package includes a repeater called DbEngPrx (dbengprx.exe).

This section includes:

Activating a Repeater

Using a Repeater

Repeater Examples

Activating a Repeater

1/3/2019 • 7 minutes to read • Edit Online

To activate the repeater connection, you will usually first start the server, then start the repeater, then start the client.

It is also possible to start the repeater first and then the server. But unless you are using the **clicon** parameter to establish a reverse connection, the client must always be started last.

Step One: Starting the Server

The server can be a debugging server, a process server, or a KD connection server. You start this as you normally would, except that the transport protocol settings will be used to connect to the repeater, not the client. For details, see **Activating a Debugging Server**, **Activating a Process Server**, or **Activating a KD Connection Server**.

If you use a password when creating the server, this password will be required when the client attaches, but not when the repeater is created.

If you use the **hidden** parameter, the server will be hidden as usual. The repeater itself is always hidden.

Step Two: Starting the Repeater

The repeater that is included in Debugging Tools for Windows is called DbEngPrx (dbengprx.exe).

DbEngPrx understands the following transport protocols: named pipe (NPIPE), TCP, and COM port.

If your client and server are using secure sockets layer (SSL) protocol, you should use TCP protocol for the repeater. If your client and server are using secure pipe (SPIPE) protocol, you should use NPIPE protocol for the repeater. The repeater will pass on whatever data it receives -- it does not interpret, encrypt, or decrypt any information. All encryption and decryption will be done by the client and the server.

The syntax for the DbEnPrx command line is as follows:

dbengprx [-p] -c *ClientTransport* **-s** *ServerTransport*

The parameters in the previous commands have the following possible values:

-p
Causes DbEngPrx to continue existing even after all connections to it are dropped.

ClientTransport
Specifies the protocol settings to be used in connecting to the server. The protocol should match that used when the server was created. The protocol syntaxes are as follows:

```
npipe:server=Server,pipe=PipeName[,password=Password]
tcp:server=Server,port=Socket[,password=Password][,ipversion=6]
tcp:clicon=Server,port=Socket[,password=Password][,ipversion=6]
com:port=COMPort,baud=BaudRate,channel=COMChannel[,password=Password]
```

The protocol parameters have the following meanings:

Server
This is the network name or IP address of the computer on which the server was created. The two initial backslashes (\) are optional.

pipe= *PipeName*
If NPIPE or SPIPE protocol is used, *PipeName* is the name that was given to the pipe when the server was created.

port= *Socket*
If TCP or SSL protocol is used, *Socket* is the same socket port number that was used when the server was created.

clicon
Specifies that the server will try to connect to the repeater through a reverse connection. *ClientTransport* must use **clicon** if and only if the server is using **clicon**. In most cases, the repeater is started before the server when a reverse connection is used.

port= *COMPort*
If COM protocol is used, *COMPort* specifies the COM port to be used. The prefix "COM" is optional -- for example, both "com2" and "2" are acceptable.

baud= *BaudRate*
If COM protocol is used, *BaudRate* should match the baud rate chosen when the server was created.

channel= *COMChannel*
If COM protocol is used, *COMChannel* should match the channel number chosen when the server was created.

password= *Password*
If a password was used when the server was created, *Password* must be supplied in order to create the debugging client. It must match the original password. Passwords are case-sensitive. If the wrong password is supplied, the error message will specify "Error 0x80004005."

ipversion=6
(Debugging Tools for Windows 6.6.07 and earlier only) Forces the debugger to use IP version 6 rather than version 4 when using TCP to connect to the Internet. In Windows Vista and later versions, the debugger attempts to auto-default to IP version 6, making this option unnecessary.

ServerTransport
Specifies the protocol settings that will be used when the client connects to the repeater. The possible protocol syntaxes are:

```
npipe:pipe=PipeName[,hidden][,password=Password][,IcfEnable]
tcp:port=Socket[,hidden][,password=Password][,IcfEnable]
tcp:port=Socket,clicon=Client[,password=Password]
com:port=COMPort,baud=BaudRate,channel=COMChannel[,hidden][,password=Password]
```

The protocol parameters have the following meanings:

pipe= *PipeName*
When NPIPE or SPIPE protocol is used, *PipeName* is a string that will serve as the name of the pipe. Each pipe name should identify a unique repeater. If you attempt to reuse a pipe name, you will receive an error message. *PipeName* must not contain spaces or quotation marks. *PipeName* can include a numerical **printf**-style format code, such as **%x** or **%d**. The repeater will replace this with the process ID of DbEngPrx. A second such code will be replaced with the thread ID of DbEngPrx.

port= *Socket*
When TCP or SSL protocol is used, *Socket* is the socket port number.

It is also possible to specify a range of ports separated by a colon. DbEngPrx will check each port in this range to see if it is free. If it finds a free port and no error occurs, the repeater will be created. The client will have to specify the actual port being used to connect to the repeater. To determine the actual port, search for the repeater; when

this repeater is displayed, the port will be followed by two numbers separated by a colon. The first number will be the actual port used; the second can be ignored. For example, if the port was specified as **port=51:60**, and port 53 was actually used, the search results will show "port=53:60". (If you are using the **clicon** parameter to establish a reverse connection, the client can specify a range of ports in this manner, while the repeater must specify the actual port used.)

clicon=*Client*
When TCP or SSL protocol is used and the **clicon** parameter is specified, a *reverse connection* will be opened. This means that the repeater will try to connect to the client, instead of letting the client initiate the contact. This can be useful if you have a firewall that is preventing a connection in the usual direction. *Client* specifies the network name or IP address of the computer on which the client exists or will be created. The two initial backslashes (\) are optional.

Since the repeater is looking for one specific client, you cannot connect multiple clients to the repeater if you use this method. If the connection is refused or is broken you will have to restart the repeater.

When **clicon** is used, it is best to start the client before the repeater is created, although the usual order (repeater before client) is also permitted.

port= *COMPort*
When COM protocol is used, *COMPort* specifies the COM port to be used. The prefix "COM" is optional -- for example, both "com2" and "2" are acceptable. You cannot use the same COM port in the *ClientTransport* and the *ServerTransport*.

baud= *BaudRate*
When COM protocol is used, *BaudRate* specifies the baud rate at which the connection will run. Any baud rate that is supported by the hardware is permitted. If you are using COM protocol in both the *ClientTransport* and the *ServerTransport* you may specify different baud rates, but naturally the slower rate will be the limit on how fast the client and server can communicate with each other.

channel= *COMChannel*
If COM protocol is used, *COMChannel* specifies the COM channel to be used in communicating with the client. This can be any value between 0 and 254, inclusive. You can use a single COM port for multiple connections using different channel numbers. (This is different from the use of a COM ports for a debug cable -- in that situation you cannot use channels within a COM port.)

hidden
Prevents the server from appearing when another debugger displays all active servers.

password= *Password*
Requires a client to supply the specified password in order to connect to the debugging session. *Password* can be any alphanumeric string.

IcfEnable
Causes the debugger to enable the necessary port connections for TCP or named pipe communication when the Internet Connection Firewall is active. By default, the Internet Connection Firewall disables the ports used by these protocols. When **IcfEnable** is used with a TCP connection, the debugger causes Windows to open the port specified by the *Socket* parameter. When **IcfEnable** is used with a named pipe connection, the debugger causes Windows to open the ports used for named pipes (ports 139 and 445). The debugger does not close these ports after the connection terminates.

Step Three: Starting the Client

The client should be a debugging client or a smart client -- whichever corresponds to your server type. For details, see **Activating a Debugging Client**, **Activating a Smart Client**, or **Activating a Smart Client (Kernel Mode)**.

If the server rejects the connection (for example, if you supply an incorrect password), both the repeater and the

client will be shut down. You will have to restart both of them to reestablish contact with the server.

Using a Repeater

12/5/2018 • 2 minutes to read • Edit Online

A repeater connection obeys very simple rules:

- Any communication that the server and client intend for each other passes through the repeater without alteration.
- Any action that the server performs with respect to the transport connection affects the repeater (and only indirectly affects the client).
- Any action that the client performs with respect to the transport connection affects the repeater (and only indirectly affects the server).

This means that any debugging commands, debugger output, control keys, and file access will take place exactly as if the client and server were directly connected. The repeater will be invisible to all these commands.

Actions that terminate the connection itself will affect the repeater. For example, if you issue a **qq (Quit)** command from the client, the server will shut down and will send a shutdown signal to the transport. This will cause the repeater to exit (unless it was started with the **-p** option). As another example, the **.clients (List Debugging Clients)** command will list the client's computer name, but it will show the connection protocol used to connect the server with the repeater.

If the server is shut down, the repeater will automatically exit (unless it was started with the **-p** option). When the repeater shuts down, this will cause a debugging client to exit as well, although a smart client will not. If for some reason you need to terminate the repeater directly, you can use Task Manager or the kill.exe tool.

Repeater Examples

12/5/2018 • 2 minutes to read • Edit Online

Let us suppose you have three computers, \\BOXA, \\BOXB, and \\BOXC, and you wish to use them as the server, the repeater, and the client, respectively.

You can start a debugging server on \\BOXA, using process 122 as the target, in the following manner:

```
E:\Debugging Tools for Windows> cdb -server tcp:port=1025,password=wrought -p 122
```

Then you can start a repeater on \\BOXB as follows:

```
C:\Misc> dbengprx -c tcp:server=BOXA,port=1025 -s npipe:pipe=MyPipe
```

Finally, start a debugging client on \\BOXC in the following manner:

```
G:\Debugging Tools> windbg -remote npipe:server=BOXB,pipe=MyPipe,password=wrought
```

Here is another example. Your symbols are at the remote location, 127.0.0.30. So you decide to use a process server on the computer where the target is, 127.0.0.10. You put a repeater at 127.0.0.20.

You also decide to use reverse connections. So you begin by starting the client on 127.0.0.30:

```
G:\Debugging Tools> windbg -premote tcp:clicon=127.0.0.20,port=1033 notepad.exe
```

Then start the repeater on 127.0.0.20:

```
C:\Misc> dbengprx -c tcp:clicon=127.0.0.10,port=1025 -s tcp:port=1033,clicon=127.0.0.10
```

And finally start the process server on 127.0.0.10:

```
E:\Debugging Tools for Windows> dbgsrv -t tcp:port=1025,clicon=127.0.0.20
```

For a more complicated example using repeaters, see Two Firewalls.

Advanced Remote Debugging Scenarios

12/5/2018 • 2 minutes to read • Edit Online

This section includes:

Debugging Targets on Multiple Computers

Symbols in the Middle

Two Firewalls

Debugging Targets on Multiple Computers

12/5/2018 • 2 minutes to read • Edit Online

The debugger can debug multiple dump files or live user-mode applications at the same time. See Debugging Multiple Targets for details.

You can debug multiple live targets even if the processes are on different systems. Simply start a process server on each system, and then use the **-premote** parameter with **.attach (Attach to Process)** or **.create (Create Process)** to identify the proper process server.

The *current* or *active* system is the system currently being debugged. If you use the **.attach** or **.create** command again without specifying the **-premote** parameter, the debugger will attach to, or create, a process on the current system.

For details on how to control such a debugging session, see Debugging Multiple Targets.

Symbols in the Middle

12/5/2018 • 2 minutes to read • Edit Online

In this scenario, you have three computers. The first has the target application, the second has the symbols, and the third has the technician.

Since the smart client behaves like a regular debugger in every way, it can be used as a debugging server at the same time. This allows you to link three machines together with the smart client in the middle.

First, you start a process server on the computer \\BOXA:

```
dbgsrv -t npipe:pipe=FarPipe
```

The middle machine, named \\BOXB, starts the debugger with both the **-premote** and **-server** parameters. Suppose the PID of the target application is 400 and the symbol path is G:\MySymbols:

```
cdb -server npipe:pipe=NearPipe -premote npipe:server=BOXA,pipe=FarPipe -v -y g:\mysymbols -p 400
```

Then a debugging client on a third machine can be started as follows:

```
windbg -remote npipe:server=BOXB,pipe=NearPipe
```

The third machine is then used to control the debugging, while the second machine is where the actual processing is done and the symbols are accessed.

Two Firewalls

12/5/2018 • 2 minutes to read • Edit Online

In this scenario, you need to perform kernel debugging on a computer in Building A. Your technician is located in Building C, and he or she has access to symbols there. However, both buildings have firewalls that will not allow incoming connections.

You need to set up a repeater at a neutral site -- say, Building B. Then you can connect A outward to B, and connect C outward to B.

There will be four computers involved in this scenario:

- The target computer, located in Building A.
- The local host computer, located in Building A. This computer will run a KD connection server. It will be connected to the target computer by a debug cable or 1394 cable, and will connect outward to the repeater. Let this computer's IP address be 127.0.10.10.
- The computer in Building B. This will run the repeater. Let its IP address be 127.0.20.20.
- The computer in Building C where the technician is located. This computer will run WinDbg as a smart client. Let its IP address be 127.0.30.30.

First, make sure the target computer is configured for debugging and is attached to the local host computer. In this example, a 1394 cable is used.

Second, start the repeater on 127.0.20.20:

```
dbengprx -p -s tcp:port=9001 -c tcp:port=9000,clicon=127.0.10.10
```

Third, start the KD connection server on 127.0.10.10 in Building A as follows:

```
kdsrv -t tcp:port=9000,clicon=127.0.20.20,password=longjump
```

Finally, start the smart client on 127.0.30.30 in Building C. (This can actually be done before or after starting the server in Building A.)

```
windbg -k kdsrv:server=@{tcp:server=127.0.20.20,port=9001,password=longjump},trans=@{1394:channel=9} -y
SymbolPath
```

Five-Computer Scenario

This scenario can be made even more complicated if you suppose that the symbols are on one computer in Building C, but the technician is at a different computer.

Suppose that 127.0.30.30 has the symbols, as before, and that its local name is \\BOXC. The smart client can be started with the same command as above but with an additional **-server** parameter. Since no one will be using this machine, it will take less processing time if you use KD instead of WinDbg:

```
kd -server npipe:pipe=randomname -k
kdsrv:server=@{tcp:server=127.0.20.20,port=9001,password=longjump},trans=@{1394:channel=9} -y SymbolPath
```

Then the technician, elsewhere in the building, can start a debugging client as follows:

```
windbg -remote npipe:server=\\BOXC,pipe=randomname
```

Notice that the password must be supplied by the first non-repeater in the chain (the smart client on \\BOXC), not by the final debugger in the chain.

Remote Debugging on Workgroup Computers

12/5/2018 • 2 minutes to read • Edit Online

You can perform remote debugging with computers that are joined to a workgroup. First configure the computer that will run the debugging server. Then activate the debugging server. After the debugging server is activated, you can connect to the server from a debugging client.

Configuring the debugging server computer

- Create a local administrator account, and log on using that account.
- Enable file and printer sharing for your active network. For example if your active network is Private, enable file and printer sharing for Private networks.

  You can use Control Panel to enable file and printer sharing. For example, here are the steps in Windows 8.

  1. Open Control Panel.
  2. Click **Network and Internet** and then **Network and Sharing Center**. Under **View your active networks**, note the type of network (for example, Private) that is active.
  3. Click **Change advanced sharing settings**. For your active network type, select **Turn on network discovery** and **Turn on file and printer sharing**.
- Start the remote registry service by following these steps.

  1. In a Command Prompt window or in the Run box, enter **services.msc**.
  2. Right click **Remote Registry**, and choose **Start**.
- Turn off the ForceGuest feature by following these steps.

  1. In a Command Prompt window or in the Run box, enter **regedit**.
  2. In Registry Editor, set this value to 0.

     HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Lsa ForceGuest

Activating the debugging server

You can activate the debugging server through the debugger or by using a process server or a KD connection server. For more information, see the following topics.

- **Activating a Debugging Server**
- **Activating a Process Server**
- **Activating a KD Connection Server**

Activating the debugging client

There are several ways to activate a debugging client. For more information, see the following topics.

- **Activating a Debugging Client**
- **Activating a Smart Client**
- **Activating a Smart Client (Kernel Mode)**
- **Searching for Process Servers**
- **Searching for KD Connection Servers**

Note If you are using a named pipe to connect a debugging client to a debugging server, you must provide the user name and password of an account that has access to the computer running the debugging server. Use one, but not both, of the following options.

- Log on to the debugging client computer with an account that shares the user name and password of an account on the debugging server computer.
- On the debugging client computer, in a Command Prompt window, enter the following command.

  net use ***Server*\ipc$ /user:***UserName*

  where *Server* is the name of the server computer, and *UserName* is the name of an account that has access to the server computer.

  When you are prompted, enter the password for *UserName*.

Debugging Previous Versions of Windows

12/5/2018 • 2 minutes to read • Edit Online

For more information about debugging with previous versions of Windows, see these topics:

- Debugging Tools For Windows: What's New
- Debugging Tools For Windows8 Release Notes
- Debugging Windows Vista
- Debugging Using Visual Studio
- Setting Up Kernel-Mode Debugging in Visual Studio

Windows 7 Debugging Tools for Windows

To debug code running on Windows Vistaor Windows Server 2008, get the Windows 7 Debugging Tools for Windows package, which is included in the Microsoft Windows Software Development Kit (SDK) for Windows 7 and .NET Framework 4.0. If you want to download only Debugging Tools for Windows, install the SDK, and, during the installation, select the **Debugging Tools for Windows** box and clear all the other boxes.

Related topics

Debugging Tools for Windows (WinDbg, KD, CDB, NTSD)

Debugging Tools For Windows: What's New

12/5/2018 • 2 minutes to read • Edit Online

In this section

- Debugging Tools for Windows: New for Windows 8.1
- Debugging Tools for Windows: New for Windows 8

Related topics

Debugging Tools for Windows (WinDbg, KD, CDB, NTSD)

Debugging Tools for Windows: New for Windows 8.1

12/5/2018 • 2 minutes to read • Edit Online

For Windows 8.1, Debugging Tools for Windows includes these new features.

- GPIO Extensions
- HID Extensions
- Most of the Kernel-Mode Driver Framework extension commands now work with UMDF 2 as well as KMDF. Some commands (for example **!wdfkd.wdfumdevstacks**) that specifically support UMDF 2 have been added to this set.
- Paging file can now be included in a CAB file along with a memory dump file. See CAB Files that Contain Paging Files Along with a Memory Dump

Related topics

Windows Debugging

Debugging Tools for Windows: New for Windows 8

12/5/2018 • 2 minutes to read • Edit Online

Beginning with Windows 8, you can develop, build, and debug both kernel-mode and user-mode components all from Microsoft Visual Studio. There is added support for debugging over a network connection or a USB 3.0 connection and improved support for debugging optimized code and inline functions. For more information, see these topics:

- Debugging Using Visual Studio
- Setting Up a Network Connection Manually
- Setting Up a USB 3.0 Connection Manually
- Debugging Optimized Code and Inline Functions

Two new sets of debugger extension commands are included:

- USB 3.0 Extensions
- RCDRKD Extensions

In addition to Visual Studio, you can sse the Windows debugger to debug Windows apps. The Debugging Tools for Windows package includes, **PLMDebug.exe**, that enables you to take manual control of suspending, resuming, debugging, and terminating Windows app.

Sos.dll is a component that is used for debugging managed code. The Windows 8 Debugging Tools for Windows package does not include sos.dll. For information about how to get sos.dll, see Getting the SOS Debugging Extension (sos.dll).

Related topics

Windows Debugging

Debugging Tools For Windows8 Release Notes

12/5/2018 • 2 minutes to read • Edit Online

Channel number for 1394 debugging in Visual Studio

If you use Microsoft Visual Studio to perform kernel-mode debugging over a 1394 cable, set the channel to a decimal integer between 1 and 62 inclusive. Do not set the channel to 0 when you first set up debugging. Because the default channel value is 0, the software assumes there is no change and does not update the settings. If you must use channel 0, first use an alternate channel (1 through 62) and then switch to channel 0.

Inline function debugging is on by default

In Windows 8, debugging of inline functions is turned on by default. The command **.inline 0** turns off inline function debugging, and the command **.inline 1** turns on inline function debugging.

Invalid port number in configuration page for network debugging

Issue: If an invalid port number is entered in the configuration page for kernel-mode network debugging, the configuration succeeds but the debugger on the host computer cannot connect to the target computer.

Workaround: Make sure the port number is valid. Valid port numbers range from 49152–65535. Also, your company might have restrictions on which ports can be used for network debugging. To ensure that a valid port number is entered, please check your internal IP Security Policy.

Use of .remote tool in command line

Issue: The use of **.remote** tool in the command line crashes the interface as it creates old style remote.exe using npipe.

Workaround: Use the **.server** command instead.

Design features for Visual Studio

- Automatic provisioning of a machine includes both user mode and kernel mode bootstrapping, regardless of which one is chosen. This requires two restarts for provisioning and takes between 8–20 minutes.
- Support for attaching only one process at a time.
- During a debugging session in Windows Debugger Extension for Visual Studio, exceptions are managed using the command line.

Global design features

- User must run in an elevated context in order to install the USB 3.0 XHCI filter driver. If the user is not running elevated, the PnP manager returns an error message that does not inform the user that elevation is the problem.
- If kernel debugging is enabled, the device used for kernel debugging should not be removed from the system while the device is still turned on. If the device is removed, the system will hang and will need to be restarted.

Debugging Windows Vista

6/25/2019 • 2 minutes to read • Edit Online

To use WinDbg to debug Windows Vista, get the Windows 7 Debugging Tools for Windows package, which is included in the Microsoft Windows Software Development Kit (SDK) for Windows 7 and .NET Framework 4.0.

If you want to download only Debugging Tools for Windows, install the SDK, and, during the installation, select the **Debugging Tools for Windows** box and clear all the other boxes.

Note You might have to uninstall Microsoft Visual C++ 2010 Redistributable components before you install the SDK. For more information, see the Microsoft Support website.

Debugging Tools (WinDbg, KD, CDB, NTSD) for Windows Vista

The Windows 7 Debugging Tools for Windows can run on x86-based or x64-based processors, and they can debug code that's running on x86-based or x64-based processors. Sometimes the debugger and the code being debugged run on the same computer, but other times the debugger and the code being debugged run on separate computers. In either case, the computer that's running the debugger is called the *host computer*, and the computer that is being debugged is called the *target computer*. Use the Windows 7 Debugging Tools for Windows when the target computer is running one of these operating systems.

Windows Vista	Windows Server 2008

If the target computer is running a more recent version of Windows, get the current Debugging Tools for Windows.

Debugging Using Visual Studio

12/5/2018 • 2 minutes to read • Edit Online

Starting with Windows Driver Kit (WDK) 8, the driver development environment and the Windows debuggers are integrated into Microsoft Visual Studio. In this integrated environment, most of the tools you need for coding, building, packaging, testing, debugging, and deploying a driver are available in the Visual Studio user interface.

> **IMPORTANT**
>
> This feature is not available in Windows 10, version 1507 and later versions of the WDK.

To get the integrated environment, first install Visual Studio, and then install the Windows Driver Kit (WDK). For more information, see Windows Driver Kit (WDK).

Typically kernel-mode debugging requires two computers. The debugger runs on the *host computer*, and the code being debugged runs on the *target computer*. The target computer is also called the *test computer*. You can do user-mode debugging on a single computer, but in some cases you might want to debug a user-mode process that is running on a separate target computer.

In the Visual Studio environment, you can configure target computers for kernel-mode and user-mode debugging. You can establish a kernel-mode debugging session. You can attach to a user-mode process or launch and debug a user mode process on the host computer or a target computer. You can analyze dump files. In Visual Studio you can sign, deploy, install, and load a driver on a target computer.

These topics show you how to use Visual Studio to perform several of the tasks involved in debugging a driver.

In this section

- Debugging a User-Mode Process Using Visual Studio
- Opening a Dump File Using Visual Studio
- Kernel-Mode Debugging in Visual Studio
- Ending a Debugging Session in Visual Studio
- Setting Symbol and Executable Image Paths in Visual Studio
- Remote Debugging Using Visual Studio
- Entering Debugger Commands in Visual Studio
- Setting Breakpoints in Visual Studio
- Viewing the Call Stack in Visual Studio
- Source Code Debugging in Visual Studio
- Viewing and Editing Memory and Registers in Visual Studio
- Controlling Threads and Processes in Visual Studio
- Configuring Exceptions and Events in Visual Studio
- Keeping a Log File in Visual Studio

Setting Up User-Mode Debugging in Visual Studio

12/5/2018 • 2 minutes to read • Edit Online

There are two user-mode debuggers available in the Microsoft Visual Studio environment. One is the Windows User-Mode Debugger, which is included in Debugging Tools for Windows. The other is the Visual Studio Debugger, which is part of the Visual Studio product. This topic describes how to get set up to use the Windows User-Mode Debugger from within the Visual Studio environment.

> **IMPORTANT**
>
> This feature is not available in Windows 10, version 1507 and later versions of the WDK.

Debugging a User-Mode Process on the Local Computer

No special setup is required for debugging a user-mode process on the local computer. For information about attaching to a process or launching a process under the debugger, see Debugging a User-Mode Process Using Visual Studio.

Debugging a User-Mode Process on a Target Computer

In some cases, two computers are used for debugging. The debugger runs on the *host computer*, and the code that is being debugged runs on the *target computer*. In Visual Studio (running on the host computer), you can use the Windows User-Mode Debugger to attach to a user-mode process on a target computer.

On the target computer, go to **Control Panel>Network and Internet>Network and Sharing Center>Advanced sharing settings**. Under **Guest or Public**, select **Turn on network discovery** and **Turn on file and printer sharing**.

You can do the rest of the configuration from the host computer:

1. On the host computer, in Visual Studio, on the **Tools** menu, choose **Attach to Process**.
2. For **Transport**, choose **Windows User Mode Debugger**.
3. To the right of the **Qualifier** box, click the **Browse** button.
4. Click the **Add** button.
5. Enter the name of the target computer.
6. To the right of **Configure Target Computer**, click the **Configure** button. The **Configure Target Computer** dialog box opens and displays the configuration progress.
7. When the configuration is complete, in the **Configure Computers** dialog box, click **OK**.

Debugging a User-Mode Process Using Visual Studio

6/25/2019 • 2 minutes to read • Edit Online

IMPORTANT

This feature is not available in Windows 10, version 1507 and later versions of the WDK.

In Microsoft Visual Studio, you can use the Windows User Mode Debugger to attach to a running process or to spawn and attach to a new process. The process can run on the same computer that is running the debugger, or it can run on a separate computer.

The procedures shown in this topic require that you have the Windows Driver Kit integrated into Visual Studio. To get the integrated environment, first install Visual Studio, and then install the Windows Driver Kit (WDK). For more information, see Windows Driver Kit (WDK).

Attaching to a running process on the same computer

1. In Visual Studio, from the **Tools** menu, choose **Attach to Process**.
2. In the **Attach to Process** dialog box, set **Transport** to **Windows User Mode Debugger**, and set **Qualifier** to **Localhost**.
3. In the **Available Processes** list, select the process that you want to attach to.
4. Click **Attach**.

Noninvasive debugging

If you want to debug a running process and interfere only minimally in its execution, you should debug the process noninvasively.

Spawning a new process on the same computer

1. In Visual Studio, from the **Tools** menu, choose **Launch Under Debugger**.
2. In the **Launch Under Debugger** dialog box, enter the path to the executable file. You can also enter arguments and a working directory.
3. Click **Launch.**

Processes that the debugger creates (also known as spawned processes) behave a little differently than processes that the debugger does not create.

Instead of using the standard heap API, processes that the debugger creates use a special debug heap. You can force a spawned process to use the standard heap instead of the debug heap by using the _NO_DEBUG_HEAP environment variable.

Also, because the target application is a child process of the debugger, it inherits the debugger's permissions. This permission might enable the target application to perform certain actions that it could not perform otherwise. For example, the target application might be able to affect protected processes.

Attaching to a running process on a separate computer

Sometimes the debugger and the code being debugged run on separate computers. The computer that runs the debugger is called the *host computer*, and the computer that runs the code being debugged is called the *target computer*. You can configure a target computer from Visual Studio on the host computer. Configuring the target

computer is also called *provisioning* the target computer. For more information, see Provision a computer for driver deployment and testing (WDK 8.1).

After you have provisioned a target computer, you can use Visual Studio on the host computer to attach to a process running on the target computer.

1. On the host computer, in Visual Studio, from the **Tools** menu, choose **Attach to Process**.
2. In the **Attach to Process** dialog box, set **Transport** to **Windows User Mode Debugger**, and set **Qualifier** to the name of the target computer.
3. In the **Available Processes** list, select the process that you want to attach to.
4. Click **Attach**.

Note If you are using separate host and target computers, do not install Visual Studio and the WDK on the target computer. Debugging is not supported if Visual Studio and the WDK are installed on the target computer.

Opening a Dump File Using Visual Studio

12/5/2018 • 2 minutes to read • Edit Online

IMPORTANT

This feature is not available in Windows 10, version 1507 and later versions of the WDK.

The procedures shown in this topic require that you have the Windows Driver Kit integrated into Visual Studio. To get the integrated environment, first install Microsoft Visual Studio, and then install the Windows Driver Kit (WDK). For more information, see Windows Driver Kit (WDK).

To open a dump file using Visual Studio:

1. In Visual Studio, from the **File** menu, choose **Open | Crash Dump** .
2. Navigate to the dump file you want to open.
3. Click **Open.**

Kernel-Mode Debugging in Visual Studio

12/5/2018 • 2 minutes to read • Edit Online

IMPORTANT

This feature is not available in Windows 10, version 1507 and later versions of the WDK.

To perform kernel-mode debugging in Microsoft Visual Studio:

1. On the host computer, in Visual Studio, from the **Tools** Menu, choose **Attach to Process**.
2. In the **Attach to Process** dialog box, set **Transport** to **Windows Kernel Mode Debugger**, and set **Qualifier** to the name of a previously configured target computer. For information about configuring a target computer, see Setting Up Kernel-Mode Debugging in Visual Studio or Setting Up Kernel-Mode Debugging Manually.
3. Click **Attach.**

Related topics

Debugging Using Visual Studio

Ending a Debugging Session in Visual Studio

12/5/2018 • 2 minutes to read • Edit Online

IMPORTANT

This feature is not available in Windows 10, version 1507 and later versions of the WDK.

To end a debugging session in Microsoft Visual Studio, from the **Debug** menu, choose **Stop Debugging**.

Setting Symbol and Executable Image Paths in Visual Studio

6/25/2019 • 2 minutes to read • Edit Online

IMPORTANT

This feature is not available in Windows 10, version 1507 and later versions of the WDK.

The procedures shown in this topic require that you have the Windows Driver Kit integrated into Visual Studio. To get the integrated environment, first install Microsoft Visual Studio, and then install the Windows Driver Kit (WDK). For more information, see Windows Driver Development.

Setting the Symbol Path by Using a Property Page

1. On the host computer, in Visual Studio, choose **Options** from the **Tools** menu.
2. In the **Options** property box, navigate to **Debugging>Symbols**.
3. If you want to get symbols from a Microsoft symbol server, check **Microsoft Symbol Servers**.
4. If you want to get symbols from folders on the host computer, check **Environment Variable: _NT_SYMBOL_PATH**. Then set the _NT_SYMBOL_PATH environment variable.

Setting the Symbol Path by Entering a Command

In Visual Studio, in the Debugger Immediate Window, enter the **.sympath (Set Symbol Path)** command.

Setting the Executable Image Path by Entering a Command

In Visual Studio, in the Debugger Immediate Window, enter the **.exepath (Set Executable Path)** command.

Remote Debugging Using Visual Studio

6/25/2019 • 2 minutes to read • Edit Online

IMPORTANT

This feature is not available in Windows 10, version 1507 and later versions of the WDK.

The procedures shown in this topic require that you have the Windows Driver Kit integrated into Visual Studio. To get the integrated environment, first install Microsoft Visual Studio, and then install the Windows Driver Kit (WDK). For more information, see Windows Driver Development.

To perform remote debugging using Visual Studio:

1. On the remote computer, in Visual Studio, choose **Connect to Remote Debugger** from the **Tools** menu.
2. In the **Connect to Remote Debugger** dialog box, enter a connection string, and click **Connect**.

Entering Debugger Commands in Visual Studio

8/26/2019 • 2 minutes to read • Edit Online

IMPORTANT

This feature is not available in Windows 10, version 1507 and later versions of the WDK. As an alternative to Visual Studio, use the command window in WinDbg Preview. For more information, see WinDbg Preview - View Menu and Debugger Commands.

The procedures shown in this topic require that you have the Windows Driver Kit integrated into Visual Studio. To get the integrated environment, first install Microsoft Visual Studio, and then install the Windows Driver Kit (WDK). For more information, see Windows Driver Development.

In Visual Studio, you can enter debugger commands in the Debugger Immediate Window. To open the Debugger Immediate Window, from the **Debug** menu, choose **Windows>Immediate**.

Setting Breakpoints in Visual Studio

6/25/2019 • 2 minutes to read • Edit Online

IMPORTANT

This feature is not available in Windows 10, version 1507 and later versions of the WDK.

The procedures shown in this topic require that you have the Windows Driver Kit integrated into Visual Studio. To get the integrated environment, first install Microsoft Visual Studio, and then install the Windows Driver Kit (WDK). For more information, see Windows Driver Development.

You can set, view, and manipulate breakpoints by entering commands in the Debugger Immediate Window. For a list of commands, see Methods of Controlling Breakpoints. If the Debugger Immediate window is not already open, from the **Debug** menu, choose **Windows>Immediate**.

Viewing the Call Stack in Visual Studio

6/25/2019 • 2 minutes to read • Edit Online

IMPORTANT

This feature is not available in Windows 10, version 1507 and later versions of the WDK.

The procedures shown in this topic require that you have the Windows Driver Kit integrated into Visual Studio. To get the integrated environment, first install Microsoft Visual Studio, and then install the Windows Driver Kit (WDK). For more information, see Windows Driver Development.

Using the Call Stack Window

To open the **Call Stack** window in Visual Studio, from the **Debug** menu, choose **Windows>Call Stack**. To set the local context to a particular row in the stack trace display, double click the first column of the row.

Viewing the Call Stack by Entering Commands

In the Debugger Immediate Window, you can view the call stack by entering one of the **k (Display Stack Backtrace)** commands. If the Debugger Immediate window is not already open, from the **Debug** menu, choose **Windows>Immediate**.

Source Code Debugging in Visual Studio

6/25/2019 • 2 minutes to read • Edit Online

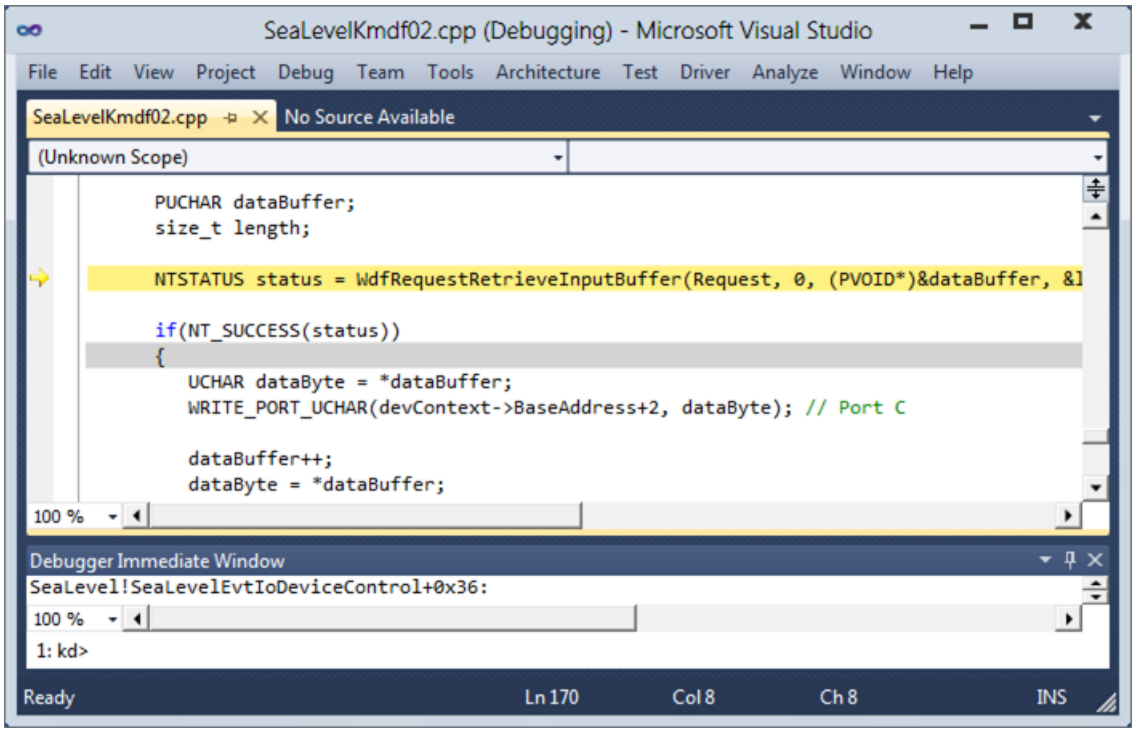
IMPORTANT

This feature is not available in Windows 10, version 1507 and later versions of the WDK.

The procedures shown in this topic require that you have the Windows Driver Kit integrated into Visual Studio. To get the integrated environment, first install Microsoft Visual Studio, and then install the Windows Driver Kit (WDK). For more information, see Windows Driver Development.

To use source debugging, you must have your compiler or linker create symbol files (.pdb files) when the binaries are built. These symbol files show the debugger how the binary instructions correspond to the source lines. Also, the debugger must be able to access the actual source files. For more information, see Source Path.

When you break in to the target computer, or when code running on the target computer hits a breakpoint, Visual Studio displays source code if it can find the source file. You can step through the source code by choosing one of the **Step** commands from the **Debug** menu. You can also set breakpoints by clicking in the left column of the source window. The following screen shot shows a source code window in the Visual Studio debugger.



Viewing and Editing Memory and Registers in Visual Studio

6/25/2019 • 2 minutes to read • Edit Online

IMPORTANT

This feature is not available in Windows 10, version 1507 and later versions of the WDK.

The procedures shown in this topic require that you have the Windows Driver Kit integrated into Visual Studio. To get the integrated environment, first install Microsoft Visual Studio, and then install the Windows Driver Kit (WDK). For more information, see Windows Driver Development.

Visual Studio provides several windows that you can use to view local variables, global variables, registers, and arbitrary memory buffers. To open any of the following windows, from the **Debug** menu, choose **Windows**.

- Locals
- Autos
- Registers
- Watch
- Memory

Controlling Threads and Processes in Visual Studio

6/25/2019 • 2 minutes to read • Edit Online

> **IMPORTANT**
> This feature is not available in Windows 10, version 1507 and later versions of the WDK.

The procedures shown in this topic require that you have the Windows Driver Kit integrated into Visual Studio. To get the integrated environment, first install Microsoft Visual Studio, and then install the Windows Driver Kit (WDK). For more information, see Windows Driver Development.

To open the Threads Window in Visual Studio, from the **Debug** menu, choose **Windows>Threads**.

Configuring Exceptions and Events in Visual Studio

6/25/2019 • 2 minutes to read • Edit Online

> **IMPORTANT**
>
> This feature is not available in Windows 10, version 1507 and later versions of the WDK.

You can configure the Windows Debugger to react to specified exceptions and events in a pre-determined way. For each exception, you can set the break status and the handling status. For each event, you can set the break status.

The procedures shown in this topic require that you have the Windows Driver Kit integrated into Visual Studio. To get the integrated environment, first install Microsoft Visual Studio, and then install the Windows Driver Kit (WDK). For more information, see Windows Driver Development.

You can configure the break status or the handling status by entering the following commands in the Debugger Immediate Window. If the Debugger Immediate Window is not already open, from the **Debug** menu, choose **Windows>Immediate**.

- **sxe**
- **sxd**
- **sxn**
- **sxi**

For a detailed discussion of exceptions and events, see Controlling Exceptions and Events.

Keeping a Log File in Visual Studio

6/25/2019 • 2 minutes to read • Edit Online

IMPORTANT

This feature is not available in Windows 10, version 1507 and later versions of the WDK.

The Windows Debugger can write a log file that records the debugging session. This log file contains all of the commands that you type and the responses from the debugger. In Microsoft Visual Studio, you can open, append, and close log files by entering commands in the Debugger Immediate Window.

The procedures shown in this topic require that you have the Windows Driver Kit integrated into Visual Studio. To get the integrated environment, first install Visual Studio, and then install the Windows Driver Kit (WDK). For more information, see Windows Driver Development.

If the Debugger Immediate Window is not already open, from the **Debug** menu, choose **Windows>Immediate**.

Opening a New Log File

To open a new log file, enter the **.logopen (Open Log File)** command. If you use the **/t** option, the date and time are appended to your specified file name. If you use the **/u** option, the log file is written in Unicode instead of in ASCII.

Appending to an Existing Log File

To append text to an existing log file, enter the **.logappend (Append Log File)** command. If you are appending to a Unicode log file, you must use the **/u** option.

Closing a Log File

To close an open log file, enter the **.logclose (Close Log File)** command.

Checking Log File Status

To determine the log file status, enter the **.logfile (Display Log File Status)** command.

Setting Up Kernel-Mode Debugging in Visual Studio

12/5/2018 • 2 minutes to read • Edit Online

IMPORTANT

This feature is not available in Windows 10, version 1507 and later versions of the WDK.

You can use Microsoft Visual Studio to set up and perform kernel-mode debugging of Windows. To use Visual Studio for kernel-mode debugging, you must have the Windows Driver Kit (WDK) integrated with Visual Studio. For information about how to install the integrated environment, see Windows Driver Development.

As an alternative to using Visual Studio to set up kernel-mode debugging, you can do the setup manually. For more information, see Setting Up Kernel-Mode Debugging Manually.

Kernel-mode debugging typically requires two computers. The debugger runs on the *host computer,* and the code being debugged runs on the *target computer*.

In this section

- Setting Up Kernel-Mode Debugging over a Network Cable in Visual Studio
- Setting Up Kernel-Mode Debugging over a 1394 Cable in Visual Studio
- Setting Up Kernel-Mode Debugging over a USB 3.0 Cable in Visual Studio
- Setting Up Kernel-Mode Debugging over a USB 2.0 Cable in Visual Studio
- Setting Up Kernel-Mode Debugging over a Serial Cable in Visual Studio
- Setting Up Kernel-Mode Debugging using Serial over USB in Visual Studio
- Setting Up Kernel-Mode Debugging of a Virtual Machine in Visual Studio

Related topics

Setting Up Debugging (Kernel-Mode and User-Mode)

Setting Up Kernel-Mode Debugging Manually

Setting Up Kernel-Mode Debugging over a Network Cable in Visual Studio

6/25/2019 • 7 minutes to read • Edit Online

IMPORTANT

This feature is not available in Windows 10, version 1507 and later versions of the WDK.

You can use Microsoft Visual Studio to set up and perform kernel-mode debugging over an Ethernet network. To use Visual Studio for kernel-mode debugging, you must have the Windows Driver Kit (WDK) integrated with Visual Studio. For information about how to install the integrated environment, see Windows Driver Development.

As an alternative to using Visual Studio to set up Ethernet debugging, you can do the setup automatically. For more information, see Setting Up KDNET Network Kernel Debugging Automatically.

Debugging over an Ethernet network has the following advantages compared to debugging over other types of cable:

- The host and target computers can be anywhere on the local network.
- It is easy to debug many target computers from one host computer.
- Network cable is inexpensive and readily available.
- Given any two computers, it is likely that they will both have Ethernet adapters. It is less likely that they will both have serial ports or both have 1394 ports.

The computer that runs the debugger is called the *host computer*, and the computer that is being debugged is called the *target computer*. The host computer must be running Windows XP or later, and the target computer must be running Windows 8 or later.

Supported network adapters

The host computer can use any wired or wireless network adapter, but the target computer must use a network adapter that is supported by Debugging Tools for Windows. For a list of supported network adapters, see Supported Ethernet NICs for Network Kernel Debugging in Windows 8.1 and Supported Ethernet NICs for Network Kernel Debugging in Windows 10.

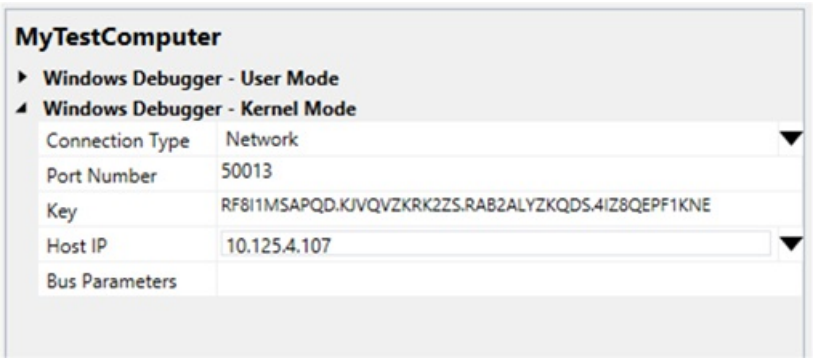
Configuring the host and target computer

1. Connect the network adapter of the target computer to a network hub or switch using and appropriate network cable. Connect the network adapter of the host computer to a network hub or switch using a standard cable or a wireless connection.

2. Begin configuring your host and target computers as described in Provision a computer for driver deployment and testing (WDK 8.1).

3. On the host computer, in Visual Studio, when you come to the Computer Configuration dialog box, select **Provision computer and choose debugger settings**.

4. For **Connection Type**, choose **Network**.



For **Port Number**, accept the default value or fill in a value of your choice. You can choose any number from 49152 through 65535. The port that you choose will be opened for exclusive access by the debugger running on the host computer. Take care to choose a port number that is not used by any other applications that run on the host computer.

Note The range of port numbers that can be used for network debugging might be limited by your company's network policy. There is no way to tell from the host computer what the limitations are. To determine whether your company's policy limits the range of ports that can be used for network debugging, check with your network administrators.

For **Key**, we strongly recommend that you use the automatically generated default value. However, you can enter your own key if you prefer. For more information, see Creating your own key later in this topic. For **Host IP**, accept the default value. This is the IP address of your host computer.

If your target computer has only one network adapter, you can leave **Bus Parameters** empty. If there is more than one network adapter on the target computer, use Device Manager on the target computer to determine the PCI bus, device, and function numbers for the adapter you want to use for debugging. For **Bus Parameters**, enter *b.d.f* where *b*, *d*, and *f* are the bus number, device number, and function number of the adapter.

5. The configuration process takes several minutes and might automatically reboot the target computer once or twice. When the process is complete, click **Finish**.

Caution If your target computer is in a docking station, and you have network debugging enabled for a network adapter that is part of the docking station, do not remove the computer from the docking station. If you need to remove the target computer from the docking station, disable kernel debugging first. To disable kernel debugging on the target computer, open a Command Prompt window as Administrator and enter the command **bcdedit /debug off**. Reboot the target computer.

Note If you intend to install the Hyper-V role on the target computer, see Setting Up Network Debugging of a Virtual Machine Host.

Verifying dbgsettings on the Target Computer

> **IMPORTANT**
> Before using BCDEdit to change boot information you may need to temporarily suspend Windows security features such as BitLocker and Secure Boot on the test PC. Re-enable these security features when testing is complete and appropriately manage the test PC, when the security features are disabled.

On the target computer, open a Command Prompt window as Administrator, and enter these commands:

bcdedit /dbgsettings

bcdedit /enum

```
...
key                     RF8...KNE
debugtype               NET
hostip                  10.125.5.10
port                    50001
dhcp                    Yes
...
busparams               0.29.7
...
```

Verify that *debugtype* is NET and *port* is the port number you specified in Visual Studio on the host computer. Also verify that *key* is the key that was automatically generated (or you specified) in Visual Studio.

If you entered **Bus Parameters** in Visual Studio, verify that *busparams* matches the bus parameters you specified.

If you do not see the value you entered for **Bus Parameters**, enter this command:

bcdedit /set "{dbgsettings}" busparams *b*.*d*.*f*

where *b*, *d*, and *f* are the bus, device, and function numbers of the network adapter on the target computer that you have chosen to use for debugging.

For example:

bcdedit /set "{dbgsettings}" busparams 0.29.7

Starting the Debugging Session

1. On the host computer, in Visual Studio, on the **Tools** menu, choose **Attach to Process**.
2. For **Transport**, choose **Windows Kernel Mode Debugger**.
3. For **Qualifier**, select the name of the target computer that you previously configured.
4. Click **Attach**.

Allowing the debugger through the firewall

When you first attempt to establish a network debugging connection, you might be prompted to allow the debugging application (Microsoft Visual Studio) through the firewall. Client versions of Windows display the prompt, but Server versions of Windows do not display the prompt. Respond to the prompt by checking the boxes for all three network types: domain, private, and public. If you do not get the prompt, or if you did not check the boxes when the prompt was available, you must use Control Panel to allow access through the firewall. Open **Control Panel > System and Security**, and click **Allow an app through Windows Firewall**. In the list of applications, use the check boxes to allow Visual Studio through the firewall. Restart Visual Studio.

Creating Your Own Key

To keep the target computer secure, packets that travel between the host and target computers must be encrypted. We strongly recommend that you use an automatically generated encryption key (provided by the Visual Studio configuration wizard) when you configure the target computer). However, you can choose to create your own key. Network debugging uses a 256-bit key that is specified as four 64-bit values, in base 36, separated by periods. Each 64-bit value is specified by using up to 13 characters. Valid characters are the letters a through z and the digits 0 through 9. Special characters are not allowed. The following list gives examples of valid (although not strong) keys:

- 1.2.3.4
- abc.123.def.456
- dont.use.previous.keys

Troubleshooting Tips for Debugging over a Network Cable

Debugging application must be allowed through firewall

Your debugger (WinDbg or KD) must have access through the firewall. You can use Control Panel to allow access through the firewall. Open **Control Panel > System and Security**, and click **Allow an app through Windows Firewall**. In the list of applications, use the check boxes to allow Visual Studio through the firewall. Restart Visual Studio.

Port number must be in range allowed by network policy

The range of port numbers that can be used for network debugging might be limited by your company's network policy. To determine whether your company's policy limits the range of ports that can be used for network debugging, check with your network administrator.

Use the following procedure if you need to change the port number.

1. On the host computer, in Visual Studio, on the **Driver** menu, choose **Test > Configure Computers**.
2. Select the name of your test computer, and click **Next**.
3. Select **Provision computer and choose debugger settings**. Click **Next**.
4. For **Port Number**, enter a number that is in the range allowed by your network administrator. Click **Next**.
5. The reconfiguration process takes a few minutes and automatically reboots the target computer. When the process is complete, click **Next** and **Finish**.

Specify busparams if target computer has multiple network adapters

If your target computer has more than one network adapter, you must specify the bus, device, and function numbers of the network adapter that you intend to use for debugging. To specify the bus parameters, open Device Manager, and locate the network adapter that you want to use for debugging. Open the property page for the network adapter, and make a note of the bus number, device number, and function number. In an elevated Command Prompt Window, enter the following command, where *b*, *d*, and *f* are the bus, device and function numbers in decimal format:

bcdedit -set "{dbgsettings}" busparams *b*.*d*.*f*

Reboot the target computer.

Related topics

Setting Up Kernel-Mode Debugging in Visual Studio

Supported Ethernet NICs for Network Kernel Debugging in Windows 10

Supported Ethernet NICs for Network Kernel Debugging in Windows 8.1

Supported Ethernet NICs for Network Kernel Debugging in Windows 8

Setting Up Kernel-Mode Debugging over a 1394 Cable in Visual Studio

6/25/2019 • 4 minutes to read • Edit Online

IMPORTANT

This feature is not available in Windows 10, version 1507 and later versions of the WDK.

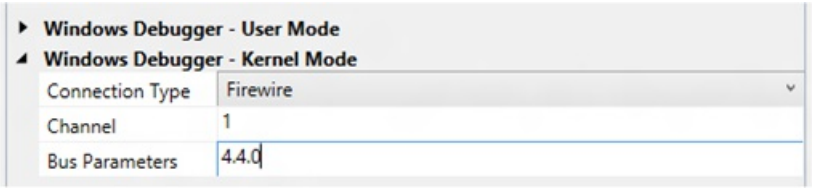
You can use Microsoft Visual Studio to set up and perform kernel-mode debugging over a 1394 (Firewire) cable. To use Visual Studio for kernel-mode debugging, you must have the Windows Driver Kit (WDK) integrated with Visual Studio. For information about how to install the integrated environment, see Windows Driver Development.

As an alternative to using Visual Studio to set up 1394 debugging, you can do the setup manually. For more information, see Setting Up Kernel-Mode Debugging over a 1394 Cable Manually.

The computer that runs the debugger is called the *host computer*, and the computer that is being debugged is called the *target computer*. The host and target computers must each have a 1394 adapter.

Configuring the host and target computers

1. Connect a 1394 cable to the 1394 controllers that you have chosen for debugging on the host and target computers.
2. Begin configuring your host and target computers as described in Provision a computer for driver deployment and testing (WDK 8.1).
3. On the host computer, in Visual Studio, when you get to the Computer Configuration dialog, select **Provision computer and choose debugger settings**.
4. For **Connection Type**, choose **Firewire**.



   For **Channel**, enter a decimal number of your choice from 1 through 62.

   Note Do not set the channel to 0 when you first set up debugging. Because the default channel value is 0, the software assumes there is no change and does not update the settings. If you must use channel 0, first use an alternate channel (1 through 62) and then switch to channel 0.

   If you have more than one 1394 controller on the target computer, enter a **Bus Parameters** value of *b.d.f*, where *b*, *d*, and *f* are the bus, device, and function numbers for the 1394 controller that you intend to use for debugging on the target computer. The bus, device, and function numbers must be in decimal format (example: 4.4.0).

5. The configuration process takes several minutes and might automatically reboot the target computer once or twice. When the process is complete, click **Finish**.

Verifying dbgsettings on the Target Computer

> **IMPORTANT**
> Before using BCDEdit to change boot information you may need to temporarily suspend Windows security features such as BitLocker and Secure Boot on the test PC. Re-enable these security features when testing is complete and appropriately manage the test PC, when the security features are disabled.

On the target computer, open a Command Prompt window as Administrator, and enter this command:

bcdedit /dbgsettings

bcdedit /enum

```
...
debugtype               1394
debugport               1
baudrate                115200
channel                 1
...
busparams               4.0.0
...
```

Verify that *debugtype* is 1394 and *channel* is the channel number you specified in Visual Studio on the host computer. You can ignore the values of *debugport* and *baudrate*; they do not apply to debugging over 1394.

If you entered **Bus Parameters** in Visual Studio, verify that *busparams* matches the bus parameters you specified.

If you do not see the value you entered for **Bus Parameters**, enter this command:

bcdedit /set "{dbgsettings}" busparams *b*.*d*.*f*

where *b*, *d*, and *f* are the bus, device, and function numbers of the 1394 controller on the target computer that you have chosen to use for debugging.

For example:

bcdedit /set "{dbgsettings}" busparams 4.4.0

Starting a Debugging Session for the First Time

1. On the host computer, open Visual Studio as Administrator.
2. On the **Tools** menu, choose **Attach to Process**.
3. For **Transport**, choose **Windows Kernel Mode Debugger**.
4. For **Qualifier**, select the name of the target computer that you previously configured.
5. Click **Attach**.

At this point, the 1394 debug driver gets installed on the host computer. This is why it is important to run Visual Studio as Administrator. After the 1394 debug driver is installed, you do not need to run as Administrator for subsequent debugging sessions.

Starting a Debugging Session

1. On the host computer, in Visual Studio, on the **Tools** menu, choose **Attach to Process**.
2. For **Transport**, choose **Windows Kernel Mode Debugger**.
3. For **Qualifier**, select the name of the target computer that you previously configured.

4. Click **Attach**.

Troubleshooting Tips for Debugging over a 1394 Cable

Most 1394 debugging problems are caused by using multiple 1394 controllers in either the host or target computer. Using multiple 1394 controllers in the host computer is not supported. The 1394 debug driver, which runs on the host, can communicate only with the first 1394 controller enumerated in the registry. If you have a 1394 controller built into the motherboard and a separate 1394 card, either remove the card or disable (by using Device Manager) the built-in controller.

The target computer can have multiple 1394 controllers, although this is not recommended. If your target computer has a 1394 controller on the motherboard, use that controller for debugging, if possible. If there is an additional 1394 card, you should remove the card and use the onboard controller. Another solution is to disable the onboard 1394 controller in the BIOS settings of the computer.

If you decide to have multiple 1394 controllers enabled on the target computer, you must specify bus parameters so that the debugger knows which controller to claim for debugging. To specify the bus parameters, Open Device Manager, and locate the 1394 controller that you want to use for debugging. Open the property page for the controller, and make a note of the bus number, device number, and function number. In an elevated Command Prompt Window, enter the following command, where *b*, *d*, and *f* are the bus, device and function numbers in decimal format:

bcdedit /set "{dbgsettings}" busparams *b*.*d*.*f*

Reboot the target computer.

Related topics

Setting Up Kernel-Mode Debugging in Visual Studio

Setting Up Kernel-Mode Debugging over a USB 3.0 Cable in Visual Studio

6/25/2019 • 5 minutes to read • Edit Online

IMPORTANT

This feature is not available in Windows 10, version 1507 and later versions of the WDK.

You can use Microsoft Visual Studio to set up and perform kernel-mode debugging over a USB 3.0 cable. To use Visual Studio for kernel-mode debugging, you must have the Windows Driver Kit (WDK) integrated with Visual Studio. For information about how to install the integrated environment, see Windows Driver Development.

As an alternative to using Visual Studio to set up USB 3.0 debugging, you can do the setup manually. For more information, see Setting Up Kernel-Mode Debugging over a USB 3.0 Cable Manually.

The computer that runs the debugger is called the *host computer*, and the computer that is being debugged is called the *target computer*.

Debugging over a USB 3.0 connection requires the following hardware:

- A USB 3.0 debug cable. This is an A-A crossover cable that has only the USB 3.0 lines and no Vbus.
- On the host computer, an xHCI (USB 3.0) host controller
- On the target computer, an xHCI (USB 3.0) host controller that supports debugging

Identifying a Debug Port on the Target Computer

1. On the target computer, launch the UsbView tool. The UsbView tool is included in Debugging Tools for Windows.
2. In UsbView, locate all of the xHCI host controllers.
3. In UsbView, expand the nodes of the xHCI host controllers. Look for an indication that a port on the host controller supports debugging.

```
[Port1]

Is Port User Connectable:         yes
Is Port Debug Capable:            yes
Companion Port Number:            3
Companion Hub Symbolic Link Name: USB#ROOT_HUB30#5&32bab638&0&0#{...}
Protocols Supported:
 USB 1.1:                         no
 USB 2.0:                         no
 USB 3.0:                         yes
```

4. Make a note of the bus, device, and function numbers for the xHCI controller that you intend to use for debugging. UsbView displays these number. In the following example, the bus number is 48, the device number is 0, and the function number is 0.

```
USB xHCI Compliant Host Controller
...
DriverKey: {36fc9e60-c465-11cf-8056-444553540000}\0020
...
Bus.Device.Function (in decimal): 48.0.0
```

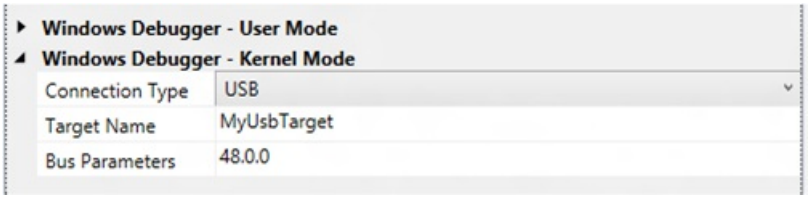
5. After you have identified an xHCI controller that supports debugging, the next step is to locate the physical USB connector that is associated with a port on the xHCI controller. To find the physical connector, plug any USB 3.0 device into any USB connector on the target computer. Refresh UsbView to see where your device is located. If UsbView shows your device connected to the xHCI host controller, then you have found a physical USB connector that you can use for USB 3.0 debugging.

Configuring the host and target computers

1. Begin configuring your host and target computers as described in Provision a computer for driver deployment and testing (WDK 8.1).

2. On the host computer, in Visual Studio, when you come to the Computer Configuration dialog box, select **Provision computer and choose debugger settings**.

3. For **Connection Type**, choose **USB**.



   For **Target Name**, enter a string to represent the target computer. This string does not have to be the official name of the target computer; it can be any string that you create as long as it meets these restrictions:

   - The maximum length of the string is 24 characters.
   - The only characters in the string are the hyphen (-), the underscore(_), the digits 0 through 9, and the letters A through Z (upper or lower case).

   If you have more than one USB host controller on the target computer, enter a **Bus Parameters** value of *b*.*d*.*f*, where *b*, *d*, and *f* are the bus, device, and function numbers for the USB host controller that you intend to use for debugging on the target computer. The bus, device, and function numbers must be in decimal format (example: 48.0.0).

4. The configuration process takes several minutes and might automatically reboot the target computer once or twice. When the process is complete, click **Finish**.

Verifying dbgsettings on the Target Computer

On the target computer, open a Command Prompt window as Administrator, and enter these commands:

bcdedit /dbgsettings

bcdedit /enum

```
...
targetname              MyUsbTarget
debugtype               USB
debugport               1
baudrate                115200
...
busparams               48.0.0
```

Verify that *debugtype* is USB and *targetname* is the name you specified in Visual Studio on the host comptuer. You can ignore the values of *debugport* and *baudrate*; they do not apply to debugging over USB.

If you entered **Bus Parameters** in Visual Studio, verify that *busparams* matches the bus parameters you specified.

If you do not see the value you entered for **Bus Parameters**, enter this command:

bcdedit /set "{dbgsettings}" busparams *b*.*d*.*f*

where *b*, *d*, and *f* are the bus, device, and function numbers of the xHCI controller on the target computer that you have chosen to use for debugging.

Example:

bcdedit /set "{dbgsettings}" busparams 48.0.0

Reboot the target computer.

Starting a Debugging Session for the First Time

1. Connect a Universal Serial Bus (USB) 3.0 debug cable to the USB 3.0 ports that you have chosen for debugging on the host and target computers.
2. On the host computer, open Visual Studio as Administrator.
3. On the **Tools** menu, choose **Attach to Process**.
4. For **Transport**, choose **Windows Kernel Mode Debugger**.
5. For **Qualifier**, select the name of the target computer that you previously configured.
6. Click **Attach**.

At this point, the USB debug driver gets installed on the host computer. This is why it is important to run Visual Studio as Administrator. After the USB debug driver is installed, you do not need to run as Administrator for subsequent debugging sessions.

Starting a Debugging Session

1. On the host computer, in Visual Studio, on the **Tools** menu, choose **Attach to Process**.
2. For **Transport**, choose **Windows Kernel Mode Debugger**.
3. For **Qualifier**, select the name of the target computer that you previously configured.
4. Click **Attach**.

Troubleshooting tips for debugging over USB 3.0

In some cases, power transitions can interfere with debugging over USB 3.0. If you have this problem, disable selective suspend for the xHCI host controller (and its root hub) that you are using for debugging.

1. In Device Manager, navigate to the node for the xHCI host controller. Right click the node, and choose **Properties**. If there is a **Power Management** tab, open the tab, and clear the **Allow the computer to turn off this device to save power** check box.
2. In Device Manager, navigate to the node for the root hub of the xHCI host controller. Right click the node, and

choose **Properties**. If there is a **Power Management** tab, open the tab, and clear the **Allow the computer to turn off this device to save power** check box.

When you have finished using the xHCI host controller for debugging, enable selective suspend for the xHCI host controller.

Related topics

Setting Up Kernel-Mode Debugging in Visual Studio

Setting Up Kernel-Mode Debugging over a USB 2.0 Cable in Visual Studio

6/25/2019 • 6 minutes to read • Edit Online

IMPORTANT

This feature is not available in Windows 10, version 1507 and later versions of the WDK.

You can use Microsoft Visual Studio to set up and perform kernel-mode debugging over a USB 2.0 cable. To use Visual Studio for kernel-mode debugging, you must have the Windows Driver Kit (WDK) integrated with Visual Studio. For information about how to install the integrated environment, see Windows Driver Development.

As an alternative to using Visual Studio to set up USB 2.0 debugging, you can do the setup manually. For more information, see Setting Up Kernel-Mode Debugging over a USB 2.0 Cable Manually.

The computer that runs the debugger is called the *host computer*, and the computer that is being debugged is called the *target computer*.

Debugging over a USB 2.0 connection requires the following hardware:

- A Universal Serial Bus (USB) 2.0 debug cable. This cable is not a standard USB 2.0 cable, because it has an extra hardware component that makes it compatible with the USB2 Debug Device Functional Specification. You can find these cables by doing an Internet search for "USB 2.0 debug cable".
- On the host computer, an EHCI (USB 2.0) host controller
- On the target computer, an EHCI (USB 2.0) host controller that supports debugging

Identifying a Debug Port on the Target Computer

1. On the target computer, launch the UsbView tool. The UsbView tool is included in Debugging Tools for Windows.
2. In UsbView, locate all of the host controllers that are compatible with the EHCI specification. For example, you could look for controllers that are listed as Enhanced.
3. In UsbView, expand the nodes of the EHCI host controllers. Look for an indication that a host controller supports debugging, and look for the number of the debug port. For example, UsbView displays this output for an EHCI host controller that supports debugging on port 1.

   ```
   Xxx xxx xxx USB2 Enhanced Host Controller - 293A
   ...
   Debug Port Number:  1
   Bus.Device.Function (in decimal): 0.29.7
   ```

   Note Many EHCI host controllers support debugging on port 1, but some EHCI host controllers support debugging on port 2.

4. Make a note of the bus, device, and function numbers for the EHCI controller that you intend to use for debugging. UsbView displays these number. In the preceding example, the bus number is 0, the device number is 29, and the function number is 7.

5. After you have identified the EHCI controller and the port number that supports debugging, the next step is to locate the physical USB connector that is associated with the correct port number. To find the physical connector, plug any USB 2.0 device into any USB connector on the target computer. Refresh UsbView to see where your device is located. If UsbView shows your device connected to the EHCI host controller and port that you identified as the debug port, then you have found a physical USB connector that you can use for debugging. It could be that there is no external physical USB connector that is associated with a debug port on an EHCI controller. In that case, you can look for a physical USB connector inside the computer. Perform the same steps to determine whether the internal USB connector is suitable for kernel debugging. If you cannot find a physical USB connector (either external or internal) that is associated with a debug port, then you cannot use the computer as a target for debugging over a USB 2.0 cable.

   Note See this remark for an exception.

Connecting the USB debug cable

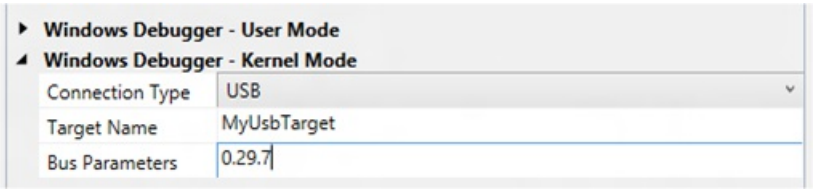
1. Verify that the host computer is not configured to be the target of USB debugging. (If necessary, open a Command Prompt window as Administrator, enter **bcdedit /debug off**, and reboot.)
2. On the host computer, use UsbView to find the EHCI host controllers and ports that support debugging. If possible, plug one end of the USB 2.0 debug cable into an EHCI port (on the host computer) that does not support debugging. Otherwise, plug the cable into any EHCI port on the host computer.
3. Plug the other end of the USB 2.0 debug cable into the connector that you identified previously on the target computer.

Configuring the host and target computers

1. Begin configuring your host and target computers as described in Provision a computer for driver deployment and testing (WDK 8.1).

2. On the host computer, in Visual Studio, when you come to the Computer Configuration dialog box, select **Provision computer and choose debugger settings**.

3. For **Connection Type**, choose **USB**.



   For **Target Name**, enter a string to represent the target computer. This string does not have to be the official name of the target computer; it can be any string that you create as long as it meets these restrictions:

   - The maximum length of the string is 24 characters.
   - The only characters in the string are the hyphen (-), the underscore(_), the digits 0 through 9, and the letters A through Z (upper or lower case).

   If you have more than one USB host controller on the target computer, enter a **Bus Parameters** value of *b*.*d*.*f*, where *b*, *d*, and *f* are the bus, device, and function numbers for the USB host controller that you intend to use for debugging on the target computer. The bus, device, and function numbers must be in decimal format (example: 0.29.7).

4. The configuration process takes several minutes and might automatically reboot the target computer once or twice. When the process is complete, click **Finish**.

Verifying dbgsettings on the Target Computer

On the target computer, open a Command Prompt window as Administrator, and enter these commands:

bcdedit /dbgsettings

bcdedit /enum

```
...
targetname              MyUsbTarget
debugtype               USB
debugport               1
baudrate                115200
...
busparams               0.29.7
...
```

Verify that *debugtype* is USB and *targetname* is the name you specified in Visual Studio on the host computer. You can ignore the values of *debugport* and *baudrate*; they do not apply to debugging over USB.

If you entered **Bus Parameters** in Visual Studio, verify that *busparams* matches the bus parameters you specified.

If you do not see the value you entered for **Bus Parameters**, enter this command:

bcdedit /set "{dbgsettings}" busparams *b*.*d*.*f*

where *b*, *d*, and *f* are the bus, device, and function numbers of the EHCI controller on the target computer that you have chosen to use for debugging.

Example:

bcdedit /set "{dbgsettings}" busparams 0.29.7

Reboot the target computer.

Starting a Debugging Session for the First Time

1. Connect a USB 2.0 debug cable to the USB 2.0 ports that you have chosen for debugging on the host and target computers.
2. On the host computer, open Visual Studio as Administrator.
3. On the **Tools** menu, choose **Attach to Process**.
4. For **Transport**, choose **Windows Kernel Mode Debugger**.
5. For **Qualifier**, select the name of the target computer that you previously configured.
6. Click **Attach**.

At this point, the USB debug driver gets installed on the host computer. This is why it is important to run Visual Studio as Administrator. After the USB debug driver is installed, you do not need to run as Administrator for subsequent debugging sessions.

Starting a Debugging Session

1. On the host computer, in Visual Studio, on the **Tools** menu, choose **Attach to Process**.
2. For **Transport**, choose **Windows Kernel Mode Debugger**.
3. For **Qualifier**, select the target name that you entered during configuration.
4. Click **Attach**.

Related topics

Setting Up Kernel-Mode Debugging in Visual Studio

Setting Up Kernel-Mode Debugging over a Serial Cable in Visual Studio

6/25/2019 • 3 minutes to read • Edit Online

IMPORTANT

This feature is not available in Windows 10, version 1507 and later versions of the WDK.

You can use Microsoft Visual Studio to set up and perform kernel-mode debugging over a null-modem cable. Null-modem cables are serial cables that have been configured to send data between two serial ports. They are available at most computer stores. Do not confuse null-modem cables with standard serial cables. Standard serial cables do not connect serial ports to each other. For information about how null-modem cables are wired, see Null-Modem Cable Wiring.

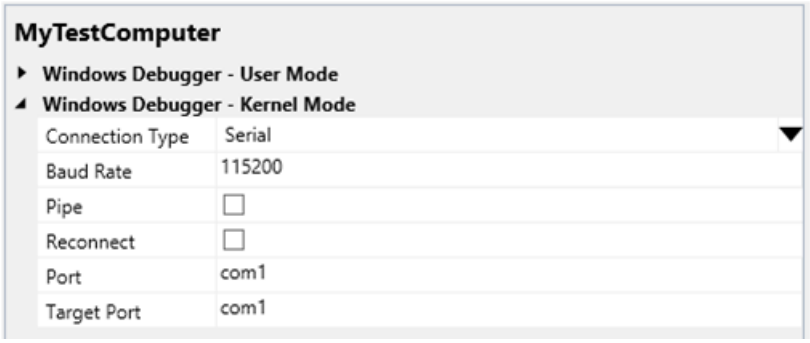
To use Visual Studio for kernel-mode debugging, you must have the Windows Driver Kit (WDK) integrated with Visual Studio. For information about how to install the integrated environment, see Windows Driver Development.

As an alternative to using Visual Studio to set up serial debugging, you can do the setup manually. For more information, see Setting Up Kernel-Mode Debugging over a Serial Cable Manually.

The computer that runs the debugger is called the *host computer*, and the computer being debugged is called the *target computer*.

Configuring the host and target computers

1. Begin configuring your host and target computer as described in Provision a computer for driver deployment and testing (WDK 8.1).
2. On the host computer, in Visual Studio, when you come to the Computer Configuration dialog box, select **Provision computer and choose debugger settings**.
3. For **Connection Type**, choose **Serial**.



   For **Baud Rate**, accept the default value or enter the baud rate you want to use. For **Port**, enter the name of the COM port that you want to use for debugging on the host computer. For **Target Port**, enter the name of the COM port that you want to use for debugging on the target computer.

4. The configuration process takes several minutes and might automatically reboot the target computer once or twice. When the process is complete, click **Finish**.

Starting the Debugging Session

1. Connect the null-modem cable to the COM ports that you have chosen for debugging on the host and target computers.
2. On the host computer, in Visual Studio, on the **Tools** menu, choose **Attach to Process**.
3. For **Transport**, choose **Windows Kernel Mode Debugger**.
4. For **Qualifier**, select the name of the target computer that you previously configured.
5. Click **Attach**.

Troubleshooting Tips for Debugging over a Serial Cable

Specify correct COM ports and baud rate

Determine the numbers of the COM ports you are using for debugging on the host and target computers. For example, suppose you have your null-modem cable connected to COM1 on the host computer and COM2 on the target computer. Also suppose you have chosen a baud rate of 115200.

1. On the host computer, in Visual Studio, on the **Driver** menu, choose **Test > Configure Computers**.
2. Select the name of your test computer, and click **Next**.
3. Select **Provision computer and choose debugger settings**. Click **Next**.
4. If you are using COM1 on the host computer, for **Port**, enter com1. If you are using COM2 on the target computer, for **Target Port**, enter com2.
5. If you have chosen to use a baud rate of 115200, for **Baud Rate**, enter 115200.

You can double check the COM port and baud rate settings on the target computer by opening a Command Prompt window as Administrator, and entering **bcdedit /dbgsettings**. If you are using COM2 on the target computer and a baud rate of 115200, the output of **bcdedit** should show `debugport 2` and `baudrate 115200`.

Null Modem Cable Wiring

The following tables show how null-modem cables are wired.

9-pin connector

CONNECTOR 1	CONNECTOR 2	SIGNALS
2	3	Tx - Rx
3	2	Rx - Tx
7	8	RTS - CTS
8	7	CTS - RTS
4	1+6	DTR - (CD+DSR)
1+6	4	(CD+DSR) - DTR
5	5	Signal ground

25-pin connector

CONNECTOR 1	CONNECTOR 2	SIGNALS
2	3	Tx - Rx

CONNECTOR 1	CONNECTOR 2	SIGNALS
3	2	Rx - Tx
4	5	RTS - CTS
5	4	CTS - RTS
6	20	DSR - DTR
20	6	DTR - DSR
7	7	Signal ground

Signal Abbreviations

ABBREVIATION	SIGNAL
Tx	Transmit data
Rx	Receive data
RTS	Request to send
CTS	Clear to send
DTR	Data terminal ready
DSR	Data set ready
CD	Carrier detect

Related topics

Setting Up Kernel-Mode Debugging in Visual Studio

Setting Up Kernel-Mode Debugging using Serial over USB in Visual Studio

6/25/2019 • 3 minutes to read • Edit Online

IMPORTANT

This feature is not available in Windows 10, version 1507 and later versions of the WDK.

The Sharks Cove development board supports serial debugging over a USB cable.

To use Microsoft Visual Studio for kernel-mode debugging, you must have the Windows Driver Kit (WDK) integrated with Visual Studio. For information about how to install the integrated environment, see Windows Driver Kit (WDK).

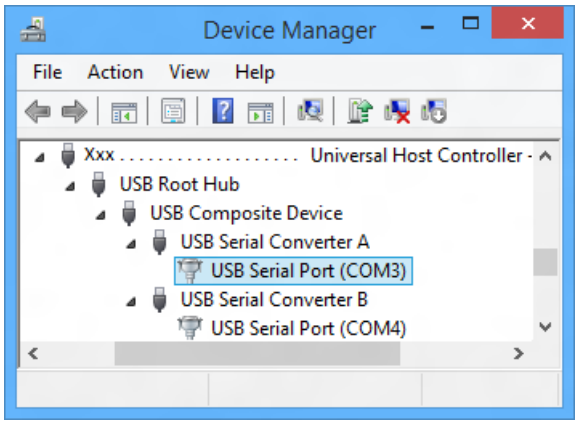
As an alternative to using Visual Studio to set up serial debugging over a USB cable, you can do the setup manually. For more information, see Setting Up Kernel-Mode Debugging using Serial over USB Manually.

The computer that runs the debugger is called the *host computer*, and the computer being debugged is called the *target computer*. In this topic, the Sharks Cove board is the target computer.

Setting up a Host Computer for debugging the Sharks Cove board

1. On the host computer, open Device Manager. On the **View** menu, choose **Devices by Type**.
2. On the Sharks Cove board, locate the debug connector. This is the micro USB connector shown in the following picture.

3. Use a USB 2.0 cable to connect the host computer to the debug connector on the Sharks cove board.
4. On the host computer, in Device Manager, two COM ports will get enumerated. Select the lowest numbered COM port. On the **View** menu, choose **Devices by Connection**. Verify that the COM port is listed under one of the USB host controllers.



Make a note of the COM port number. This is the lowest COM port number that shows under the USB host controller node. For example, in the preceding screen shot, the lowest COM port number under the USB host controller is COM3. You will need this COM port number later when you start a debugging session. If the driver is not already installed for the COM port, right click the COM port node, and choose **Update Driver**. Then select **Search automatically for updated driver software**. You will need an internet connection for this.

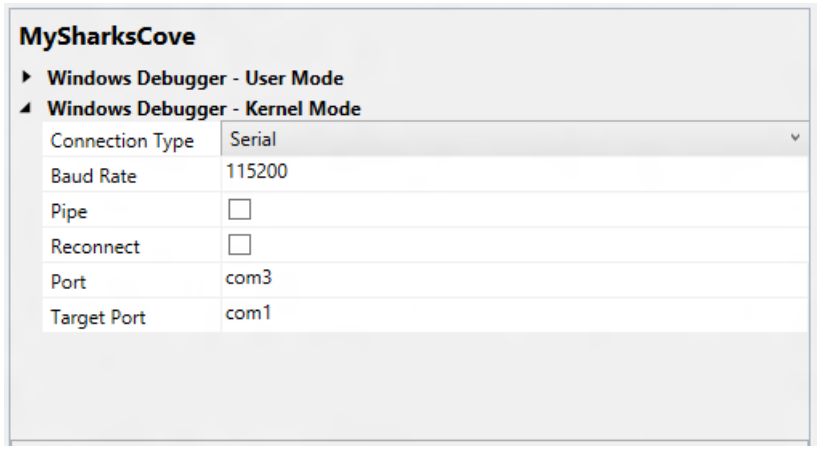
Configuring the host and target computers

In these steps, the Sharks Cove board is the target computer.

1. Begin configuring your host and target computer as described in Provision a computer for driver deployment and testing (WDK 8.1).

2. On the host computer, in Visual Studio, when you come to the Computer Configuration dialog box, select **Provision computer and choose debugger settings**.

3. For **Connection Type**, choose **Serial**.



For **Baud Rate**, enter 115200. For **Port**, enter the name of the COM port that you noted previously in Device Manager (for example, com3). For **Target Port**, enter com1.

4. The configuration process takes several minutes and might automatically reboot the target computer once or twice. When the process is complete, click **Finish**.

Starting the Debugging Session

1. On the host computer, in Visual Studio, on the **Tools** menu, choose **Attach to Process**.
2. For **Transport**, choose **Windows Kernel Mode Debugger**.
3. For **Qualifier**, select the name of the target computer that you previously configured.
4. Click **Attach**.

Troubleshooting Tips for Serial Debugging over a USB Cable

Specify correct COM port on both host and target

On the target computer (Sharks Cove board), verify that you are using COM1 for debugging. Open a Command Prompt window as Administrator, and enter **bcdedit /dbgsettings**. The output of **bcdedit** should show `debugport 1`.

On the host computer, verify that you are using the COM port that you noted earlier in Device Manager.

1. On the host computer, in Visual Studio, on the **Driver** menu, choose **Test > Configure Computers**.
2. Select the name of your test computer, and click **Next**.
3. Select **Provision computer and choose debugger settings**. Click **Next**.
4. Verify that the correct COM port number is listed for **Port**.

Baud rate must be the same on host and target

The baud rate must be 115200 on both the host and target computers.

On the target computer (Sharks Cove board), open a Command Prompt window as Administrator, and enter **bcdedit /dbgsettings**. The output of **bcdedit** should show `baudrate 115200`.

On the host computer, verify that you are using a baud rate of 115200.

1. On the host computer, in Visual Studio, on the **Driver** menu, choose **Test > Configure Computers**.
2. Select the name of your test computer, and click **Next**.
3. Select **Provision computer and choose debugger settings**. Click **Next**.
4. Verify that the **Baud Rate** is 115200.

Related topics

Setting Up Kernel-Mode Debugging in Visual Studio

Setting Up Kernel-Mode Debugging of a Virtual Machine in Visual Studio

12/5/2018 • 3 minutes to read • Edit Online

IMPORTANT

This feature is not available in Windows 10, version 1507 and later versions of the WDK.

You can use Microsoft Visual Studio to set up and perform kernel-mode debugging of a virtual machine. The virtual machine can be located on the same physical computer as the debugger or on a different computer that is connected to the same network. To use Visual Studio for kernel-mode debugging, you must have the Windows Driver Kit (WDK) integrated with Visual Studio. For information about how to install the integrated environment, see Windows Driver Development.

The computer that runs the debugger is called the *host computer*, and the virtual machine that is being debugged is called the *target virtual machine*.

Configuring the Target Virtual Machine

1. In the virtual machine, in an elevated Command Prompt window, enter the following commands.

   bcdedit /debug on

   bcdedit /dbgsettings serial debugport:*n* **baudrate:115200**

   where *n* is the number of a COM port on the virtual machine.

2. Reboot the virtual machine.

3. In the virtual machine, configure the COM port to map to a named pipe. The debugger will connect through this pipe. For more information about how to create this pipe, see your virtual machine's documentation.

Configuring the Host Computer

The host computer can be the same physical computer that is running the virtual machine, or it can be a separate computer.

1. On the host computer, in Visual Studio, on the **Driver** menu, choose **Test > Configure Computer**.

2. Click **Add New Computer**.

3. For **Computer name**, enter the name of the physical computer that is running the target virtual machine.

4. Select **Manually configure debuggers and do not provision**, and click **Next**.

5. For **Connection Type**, select **Serial**.

6. Check **Pipe**, and check **Reconnect**.

7. If the debugger is running on the same computer as the virtual machine, enter the following for **Pipe name**:

   \\.\pipe*PipeName*.

If the debugger is running on a different computer from the virtual machine, enter the following for **Pipe name**:

*****VMHost***\pipe***PipeName*

where, *VMHost* is the name of the physical computer that is running the target virtual machine, and *PipeName* is the name of the pipe that you associated with the COM port on the target virtual machine.

8. Click **Next**. Click **Finish**.

Starting the Debugging Session

1. On the host computer, in Visual Studio, on the **Debug** menu, choose **Attach to Process**.
2. For **Transport**, choose **Windows Kernel Mode Debugger**.
3. For **Qualifier**, select the name of the physical computer that is running the target virtual machine.
4. Click **Attach**.

> **TIP**
> If you receive the message "Could not start debug session, error 8007005: Access denied", run Visual Studio as ADMINISTRATOR on the host computer.

Generation 2 Virtual Machines

By default, COM ports are not presented in generation 2 virtual machines. You can add COM ports through PowerShell or WMI. For the COM ports to be displayed in the Hyper-V Manager console, they must be created with a path.

To enable kernel debugging using a COM port on a generation 2 virtual machine, follow these steps:

1. Disable Secure Boot by entering this PowerShell command:

   Set-VMFirmware –Vmname *VmName* **–EnableSecureBoot Off**

   where *VmName* is the name of your virtual machine.

2. Add a COM port to the virtual machine by entering this PowerShell command:

   Set-VMComPort –VMName *VmName* **1 \\.\pipe***PipeName*

   For example, the following command configures the first COM port on virtual machine TestVM to connect to named pipe TestPipe on the local computer.

   Set-VMComPort –VMName TestVM 1 \\.\pipe\TestPipe

3. Restart the VM so that the new settings are in effect.

For more information, see Generation 2 Virtual Machine Overview.

Troubleshooting Firewalls and Network Access Issues

Your debugger (WinDbg or KD) must have access through the firewall. This can even be the case for virtual serial ports that are supported by network adapters.

If you are prompted by Windows to turn off the firewall when the debugger is loaded, select **all three** boxes.

Depending on the specifics of the VM in use, you may need to change the network settings for your virtual machines to bridge them to the Microsoft Kernel Network Debug Adapter. Otherwise, the virtual machines will not

have access to the network.

Windows Firewall

You can use Control Panel to allow access through the Windows firewall.

1. Open Control Panel > System and Security, and select Allow an app through Windows Firewall.
2. In the list of applications, locate *Windows GUI Symbolic Debugger* and *Windows Kernel Debugger*.
3. Use the check boxes to allow those two applications through the firewall. Click OK to save your settings.
4. Restart your debugging application (WinDbg or KD).

Related topics

Setting Up Network Debugging of a Virtual Machine Host

Debugging Techniques

12/5/2018 • 2 minutes to read • Edit Online

This section discusses two types of debugging techniques: standard and specialized. Standard techniques apply to most debugging scenarios, and examples include setting breakpoints, inspecting the call stack, and finding a memory leak. Specialized techniques apply to particular technologies or types of code, and examples are Plug and Play debugging, Kernel Mode Driver Framework debugging, and RPC debugging.

You can learn more the following sections.

Standard Debugging Techniques

Specialized Debugging Techniques

Standard Debugging Techniques

12/5/2018 • 2 minutes to read • Edit Online

This section discusses standard debugging techniques that you can apply across different technologies and different types of code.

In this section

- Using Breakpoints
- Reading and Writing Memory
- Using the !analyze Extension
- Handling a Bug Check When Driver Verifier is Enabled
- Noninvasive Debugging (User Mode)
- Debugging in Assembly Mode
- Debugging in Source Mode
- Debugging Optimized Code and Inline Functions
- Debugging Managed Code Using the Windows Debugger
- Debugging Windows Apps Using the Windows Debugger
- Changing Contexts
- Controlling Processes and Threads
- Using Debugger Markup Language
- Controlling Exceptions and Events
- Finding the Process ID
- Debugging a Stack Overflow
- Manually Walking a Stack
- Debugging a Stack Trace that has JScript Frames
- Debugging an Application Failure
- Reattaching to the Target Application
- Crashing and Rebooting the Target Computer
- Synchronizing with the Target Computer
- Finding a Memory Leak
- Debugging a Time Out
- Debugging a Stalled System
- Debugging Multiple Targets
- Tracking Down a Processor Hog
- Determining the ACL of an Object
- Displaying a Critical Section
- Debugging a Deadlock
- Debugging a Failed Driver Unload
- Reading Bug Check Callback Data
- Debugging a User-Mode Failure with KD
- Crashing and Rebooting the Target Computer
- Mapping Driver Files
- Messages from the Target

Using Breakpoints - Debugging Techniques

12/5/2018 • 2 minutes to read • Edit Online

A *breakpoint* is a location in executable code at which the operating system stops execution and breaks into the debugger. This allows you to analyze the target and issue debugger commands.

This section includes the following topics:

Methods of Controlling Breakpoints

Breakpoint Syntax

Unresolved Breakpoints (bu Breakpoints)

Processor Breakpoints (ba Breakpoints)

Initial Breakpoint

User Space and System Space

Risks Entailed When Setting Breakpoints

Methods of Controlling Breakpoints

12/5/2018 • 3 minutes to read • Edit Online

You can specify the location of a breakpoint by virtual address, module and routine offsets, or source file and line number (when in source mode). If you put a breakpoint on a routine without an offset, the breakpoint is activated when that routine is entered.

There are several additional kinds of breakpoints:

- A breakpoint can be associated with a certain thread.
- A breakpoint can enable a fixed number of passes through an address before it is triggered.
- A breakpoint can automatically issue certain commands when it is triggered.
- A breakpoint can be set on non-executable memory and watch for that location to be read or written to.

If you are debugging more than one process in user mode, the collection of breakpoints depends on the current process. To view or change a process' breakpoints, you must select the process as the current process. For more information about the current process, see Controlling Processes and Threads.

Debugger Commands for Controlling and Displaying Breakpoints

To control or display breakpoints, you can use the following methods:

- Use the **bl (Breakpoint List)** command to list existing breakpoints and their current status.
- Use the **.bpcmds (Display Breakpoint Commands)** command to list all breakpoints along with the commands that were used to create them.
- Use the **bp (Set Breakpoint)** command to set a new breakpoint.
- Use the **bu (Set Unresolved Breakpoint)** command to set a new breakpoint. Breakpoints that are set with **bu** are called unresolved breakpoints; they have different characteristics than breakpoints that are set with **bp**. For complete details, see Unresolved Breakpoints (bu Breakpoints).
- Use the **bm (Set Symbol Breakpoint)** command to set new breakpoints on symbols that match a specified pattern. A breakpoint set with **bm** will be associated with an address (like a **bp** breakpoint) if the **/d** switch is included; it will be unresolved (like a **bu** breakpoint) if this switch is not included.
- Use the **ba (Break on Access)** command to set a *processor breakpoint*, also known as a *data breakpoint*. These breakpoints can be triggered when the memory location is written to, when it is read, when it is executed as code, or when kernel I/O occurs. For complete details, see Processor Breakpoints (ba Breakpoints).
- Use the **bc (Breakpoint Clear)** command to permanently remove one or more breakpoints.
- Use the **bd (Breakpoint Disable)** command to temporarily disable one or more breakpoints.
- Use the **be (Breakpoint Enable)** command to re-enable one or more disabled breakpoints.
- Use the **br (Breakpoint Renumber)** command to change the ID of an existing breakpoint.
- Use the **bs (Update Breakpoint Command)** command to change the command associated with an existing breakpoint.
- Use the **bsc (Update Conditional Breakpoint)** command to change the condition under which an

existing conditional breakpoint occurs.

In Visual Studio and WinDbg, there are several user interface elements that facilitate controlling and displaying breakpoints. See Setting Breakpoints in Visual Studio and Setting Breakpoints in WinDbg.

Each breakpoint has a decimal number called the breakpoint ID associated with it. This number identifies the breakpoint in various commands.

Breakpoint Commands

You can include a command in a breakpoint that is automatically executed when the breakpoint is hit. For example, the following command breaks at MyFunction+0x47, writes a dump file, and then resumes execution.

```
0:000> bu MyFunction+0x47 ".dump c:\mydump.dmp; g"
```

Note If you are controlling the user-mode debugger from the kernel debugger, do not use **g (Go)** in the breakpoint command string. The serial interface might be unable to keep up with this command, and you will be unable to break back into CDB. For more information about this situation, see Controlling the User-Mode Debugger from the Kernel Debugger.

Number of Breakpoints

In kernel mode, you can use a maximum of 32 software breakpoints. In user mode, you can use any number of software breakpoints.

The number of processor breakpoints that are supported depends on the target processor architecture.

Conditional Breakpoints

You can set a breakpoint that is triggered only under certain conditions. For more information about these kinds of breakpoints, see Setting a Conditional Breakpoint.

Breakpoint Syntax

7/9/2019 • 2 minutes to read • Edit Online

The following syntax elements can be used when creating a breakpoint, either through the Debugger Command window or through the WinDbg graphical interface.

Addresses in Breakpoints

Breakpoints support many kinds of address syntax, including virtual addresses, function offsets, and source line numbers. For example, you can use any of the following commands to set breakpoints:

```
0:000> bp 0040108c
0:000> bp main+5c
0:000> bp `source.c:31`
```

For more information about this syntax, see Numerical Expression Syntax, Source Line Syntax, and the individual command topics.

Breakpoints on Methods

If you want to put a breakpoint on the *MyMethod* method in the *MyClass* class, you can use two different syntaxes:

- In MASM expression syntax, you can indicate a method by a double colon or by a double underscore.

  ```
  0:000> bp MyClass::MyMethod
  0:000> bp MyClass__MyMethod
  ```

- In C++ expression syntax, you must indicate a method by a double colon.

  ```
  0:000> bp @@( MyClass::MyMethod )
  ```

If you want to use a more complex breakpoint command, you should use MASM expression syntax. For more information about expression syntax, see Evaluating Expressions.

Breakpoints Using Complicated Text

To set a breakpoint on complicated functions, including functions that contain spaces, as well as a member of a C++ public class, enclose the expression in parentheses. For example, use **bp (??MyPublic)** or **bp (operator new)**.

A more versatile technique is to use the @!"chars" syntax. This is a special escape in the MASM evaluator that enables you to provide arbitrary text for symbol resolution. You must start with the three symbols @!" and end with a quotation mark ("). Without this syntax, you cannot use spaces, angle brackets (<, >), or other special characters in symbol names in the MASM evaluator. This syntax is exclusively for names, and not parameters. Templates and overloads are the primary sources of symbols that require this quote notation. You can also set the **bu** command by using the @!"chars" syntax, as the following code example shows.

```
0:000> bu @!"ExecutableName!std::pair<unsigned int,std::basic_string<unsigned short,std::char_traits<unsigned
short>,std::allocator<unsigned short> > >::operator="
```

In this example, *ExecutableName* is the name of an executable file.

This escape syntax is more useful for C++ (for example, overloaded operators) instead of C because there are no

spaces (or special characters) in C function names. However, this syntax is also important for a lot of managed code because of the considerable use of overloads in the .NET Framework.

To set a breakpoint on arbitrary text in C++ syntax, use **bu @@c++(*text*)** for C++-compatible symbols.

Breakpoints in Scripts

Breakpoint IDs do not have to be referred to explicitly. Instead, you can use a numerical expression that resolves to an integer that corresponds to a breakpoint ID. To indicate that the expression should be interpreted as a breakpoint, use the following syntax.

```
b?[Expression]
```

In this syntax, the square brackets are required, and *Expression* stands for any numerical expression that resolves to an integer that corresponds to a breakpoint ID.

This syntax allows debugger scripts to programmatically select a breakpoint. In the following example, the breakpoint changes depending on the value of a user-defined pseudo-register.

```
b?[@$t0]
```

Breakpoint Pseudo-Registers

If you want to refer to a breakpoint address in an expression, you can use a pseudo-register with the **$bp***Number* syntax, where *Number* is the breakpoint ID. For more information about this syntax, see Pseudo-Register Syntax.

Unresolved Breakpoints (bu Breakpoints)

12/5/2018 • 2 minutes to read • Edit Online

If a breakpoint is set for a routine name that has not been loaded, the breakpoint is called a *deferred*, *virtual*, or *unresolved* breakpoint. (These terms are used interchangeably.) Unresolved breakpoints are not associated with any specific load of a module. Every time that a new application is loaded, it is checked for this routine name. If this routine appears, the debugger computes the actual coded address of the virtual breakpoint and enables the breakpoint.

If you set a breakpoint by using the **bu** command, the breakpoint is automatically considered unresolved. If this breakpoint is in a loaded module, the breakpoint is still enabled and functions normally. However, if the module is later unloaded and reloaded, this breakpoint does not vanish. On the other hand, a breakpoint that you set with **bp** is immediately resolved to an address.

There are three primary differences between **bp** breakpoints and **bu** breakpoints:

- A **bp** breakpoint location is always converted to an address. If a module change moves the code at which a **bp** breakpoint was set, the breakpoint remains at the same address. On the other hand, a **bu** breakpoint remains associated with the symbolic value (typically a symbol plus an offset) that was used, and it tracks this symbolic location even if its address changes.
- If a **bp** breakpoint address is found in a loaded module, and if that module is later unloaded, the breakpoint is removed from the breakpoint list. On the other hand, **bu** breakpoints persist after repeated unloads and loads.
- Breakpoints that you set with **bp** are not saved in WinDbg workspaces. Breakpoints that are set with **bu** are saved in workspaces.

Controlling Address Breakpoints and Unresolved Breakpoints

Address breakpoints can be created with the **bp (Set Breakpoint)** command, or the **bm (Set Symbol Breakpoint)** command when the **/d** switch is included. Unresolved breakpoints can be created with the **bu (Set Unresolved Breakpoint)** command, or the **bm** command when the **/d** switch is not included. Commands that disable, enable, and modify breakpoints apply to all kinds of breakpoints. Commands that display a list of breakpoints include all breakpoints, and indicate the type of each. For a listing of these commands, see Methods of Controlling Breakpoints.

The WinDbg **Breakpoints** dialog box displays all breakpoints, indicating unresolved breakpoints with the notation "u". This dialog box can be used to modify any breakpoint.The **Command** text box on this dialog box can be used to create any type of breakpoint; if the type is omitted, an unresolved breakpoint is created. For details, see Edit | Breakpoints. When you set a breakpoint by using the mouse in the WinDbg Disassembly window or Source window, the debugger creates an unresolved breakpoint.

Processor Breakpoints (ba Breakpoints)

12/5/2018 • 6 minutes to read • Edit Online

Breakpoints that are controlled by the processor at the request of the debugger are known as *processor breakpoints* or *data breakpoints*. Breakpoints that are controlled directly by the debugger are known as *software breakpoints*.

Note Although the term *data breakpoint* is commonly used as a synonym for *processor breakpoint*, this term can be misleading. There are two fundamental types of breakpoints: processor breakpoints, which are controlled by the processor, and software breakpoints, which are controlled by the debugger. Processor breakpoints are usually set on program data -- this is the reason they are called "data breakpoints" -- but they can also be set on executable code. Software breakpoints are usually set on executable code, but they can also be set on program data. Unfortunately, it is common in debugging literature to refer to processor breakpoints as "data breakpoints", even when they are set on executable code.

Processor Breakpoints

A processor breakpoint is triggered when a specific memory location is accessed. There are four types of processor breakpoints, corresponding to the kind of memory access that triggers it:

BREAKPOINT TYPE	ACTION
e (execute)	Triggered when the processor retrieves an instruction from the specified address.
r (read/write)	Triggered when the processor reads or writes memory at the specified address.
w (write)	Triggered when the processor writes memory at the specified address.
i (i/o)	Triggered when the I/O port at the specified *Address* is accessed.

Each processor breakpoint has a size associated with it. For example, a **w** (write) processor breakpoint could be set at the address 0x70001008 with a size of four bytes. This would monitor the block of addresses from 0x70001008 to 0x7000100B, inclusive. If this block of memory is written to, the breakpoint will be triggered.

It can happen that the processor performs an operation on a memory region that overlaps with, but is not identical to, the specified region. In the example given in the preceding paragraph, a single write operation that includes the range 0x70001000 to 0x7000100F, or a write operation that includes only the byte at 0x70001009, would be an overlapping operation. In such a situation, whether the breakpoint is triggered is processor-dependent. For details of how this situation is handled on a specific processor, consult the processor archictecture manual and look for "debug register" or "debug control register". To take one specific processor type as an example, on an x86 processor, a read or write breakpoint is triggered whenever the accessed range overlaps the breakpoint range.

Similarly, if an **e** (execute) breakpoint is set on the address 0x00401003, and then a two-byte instruction spanning the addresses 0x00401002 and 0x00401003 is executed, the result is processor-dependent. Again, consult the processor architecture manual for details.

The processor distinguishes between breakpoints set by a user-mode debugger and breakpoints set by a kernel-mode debugger. A user-mode processor breakpoint does not affect any kernel-mode processes. A kernel-mode processor breakpoint might or might not affect a user-mode process, depending on whether the user-mode code is using the debug register state and whether there is a user-mode debugger that is attached.

To apply the current process' existing data breakpoints to a different register context, use the **.apply_dbp (Apply Data Breakpoint to Context)** command.

On a multiprocessor computer, each processor breakpoint applies to all processors. For example, if the current processor is 3 and you use the command `ba e1 MyAddress` to put a breakpoint at **MyAddress**, any processor -- not only processor 3 -- that executes at that address triggers the breakpoint. This holds for software breakpoints as well.

Software Breakpoints

Software breakpoints, unlike processor breakpoints, are controlled by the debugger. When the debugger sets a software breakpoint at some location, it temporarily replaces the contents of that memory location with a break instruction. The debugger remembers the original contents of this location, so that if this memory is displayed in the debugger, the debugger will show the original contents of that memory location, not the break instruction. When the target process executes the code at this location, the break instruction causes the process to break into the debugger. After you have performed whatever actions you choose, you can cause the target to resume execution, and execution will resume with the instruction that was originally in that location.

Availability of Processor Breakpoint Types

On Windows Server 2003 with Service Pack 1 (SP1), on an Itanium-based computer that uses WOW64 to emulate x86, processor breakpoints do not work with the **e** (execute) option but they do work with the **r** (read/write) and **w** (write) options.

The **i** (i/o) option is available only during kernel-mode debugging, with a target computer that is running Windows XP or a later version of Windows on an x86-based processor.

Not all data sizes can be used with all processor breakpoint types. The permitted sizes depend on the processor of the target computer. For details, see **ba (Break on Access)**.

Limitations of Software Breakpoints and Processor Breakpoints

It is possible to specify a data address rather than a program address when using the **bp** or bm /a commands. However, even if a data location is specified, these commands create software breakpoints, not processor breakpoints. When the debugger places a software breakpoint at some location, it temporarily replaces the contents of that memory location with a break instruction. This does not corrupt the executable image, because the debugger remembers the original contents of this location, and when the target process attempts to execute this code the debugger can respond appropriately. But when a software breakpoint is set in a data location, the resulting overwrite can lead to data corruption. Therefore, setting a software breakpoint on a data location is safe only if you are certain that this location will be used only as executable code.

The **bp**, **bu**, and **bm** commands set software breakpoints by replacing the processor instruction with a break instruction. Therefore these cannot be used in read-only code or any other code that cannot be overwritten. To set a breakpoint in such code, you must use **ba (Break on Access)** with the **e** (execute) option.

You cannot create multiple processor breakpoints at the same address that differ only in the command that is automatically executed when the breakpoint is triggered. However, you can create multiple breakpoints at the same address that differ in their other restrictions (for example, you can create multiple breakpoints at the same address by using the **ba** command with different values of the **/p**, **/t**, **/c**, and **/C** options).

The initial breakpoint in a user-mode process (typically set on the **main** function or its equivalent) cannot be a processor breakpoint.

The number of processor breakpoints that are supported depends on the target processor architecture.

Controlling Software Breakpoints and Processor Breakpoints

Software breakpoints can be created with the **bp (Set Breakpoint)**, **bm (Set Symbol Breakpoint)**, and **bu (Set Unresolved Breakpoint)** commands. Processor breakpoints can be created with the **ba (Break on Access)** command. Commands that disable, enable, and modify breakpoints apply to all kinds of breakpoints. Commands that display a list of breakpoints include all breakpoints, and indicate the type of each. For a listing of these commands, see Methods of Controlling Breakpoints.

The WinDbg **Breakpoints** dialog box displays all breakpoints, indicating processor breakpoints with the notation "e", "r", "w", or "i' followed by the size of the block. This dialog box can be used to modify any breakpoint. The **Command** text box on this dialog box can be used to create any type of breakpoint.If a processor breakpoint is desired, begin the input with "ba". For details, see Edit | Breakpoints. When you set a breakpoint by using the mouse in the WinDbg Disassembly window or Source window, the debugger creates an unresolved software breakpoint.

Processor breakpoints are stored in the processor's debug registers. It is possible to set a breakpoint by manually editing a debug register value, but this is strongly discouraged.

Initial Breakpoint

12/5/2018 • 2 minutes to read • Edit Online

When the debugger starts a new target application, an initial breakpoint automatically occurs after the main image and all statically-linked DLLs are loaded before any DLL initialization routines are called.

When the debugger attaches to an existing user-mode application, an initial breakpoint occurs immediately.

The **-g** command-line option causes WinDbg or CDB to ignore the initial breakpoint. You can automatically execute a command at this point. For more information about this situation, see Controlling Exceptions and Events.

If you want to start a new target and break into it when the execution of the actual application is about to begin, do not use the **-g** option. Instead, let the initial breakpoint occur. After the debugger is active, set a breakpoint on the **main** or **winmain** routine and then use the **g (Go)** command. All of the initialization procedures then run, and the application stops when execution of the main application is about to begin.

For more information about automatic breakpoints in kernel mode, see Crashing and Rebooting the Target Computer.

User Space and System Space

12/5/2018 • 2 minutes to read • Edit Online

Windows gives each user-mode application a block of virtual addresses. This is known as the *user space* of that application. The other large block of addresses, known as *system space* or *kernel space*, cannot be directly accessed by the application.

When WinDbg or CDB sets a breakpoint in user space, this breakpoint is set at the specified address in the user space of a single process. During user-mode debugging, the current process determines the meaning of virtual addresses. For more information, see Controlling Processes and Threads.

In kernel mode, you can set breakpoints in user space with the **bp**, **bu**, and **ba** commands or with the **Breakpoints** dialog box. You must first use the *process context* to specify the user-mode process that owns that address space by using **.process /i** (or a process-specific breakpoint on some kernel-space function) to switch the target to the correct process context.

Breakpoints in user space are always associated with the process whose process context was active when the breakpoints were set. If a user-mode debugger is debugging this process and if a kernel debugger is debugging the computer that the process is running on, this breakpoint breaks into the user-mode debugger, even though the breakpoint was actually set from the kernel debugger. You can break into the system from the kernel debugger at this point, or use the **.breakin (Break to the Kernel Debugger)** command from the user-mode debugger to transfer control to the kernel debugger.

Determining the Range of User Space and System Space

If you need to determine the extent of user space and system space on the target computer, you can use the **dp (Display Memory)** command from a kernel debugger to display the Windows global variable **MmHighestUserAddress**. This variable contains the address of the top of user space. Since system space addresses are always higher than user space addresses, this value allows you to determine whether any given address is in user space or in kernel space.

For example, on a 32-bit target computer with an x86 processor and standard boot parameters, this command will show the following result:

```
kd> dp nt!mmhighestuseraddress L1
81f71864  7ffeffff
```

This indicates that user space ranges from the address 0x00000000 to 0x7FFEFFFF, and system space therefore ranges from 0x80000000 up to the highest possible address (which is 0xFFFFFFFF on a standard 32-bit Windows installation).

With a 64-bit target computer, different values will occur. For example, this command might show the following:

```
0: kd> dp nt!mmhighestuseraddress L1
fffff800`038b4010  000007ff`fffeffff
```

This indicates that user space ranges from 0x00000000`00000000 to 0x000007FF`FFFEFFFF. Therefore, system space includes all addresses from 0x00000800`00000000 upward.

For more information about Windows memory management, see *Microsoft Windows Internals* by David Solomon and Mark Russinovich (4th edition, Microsoft Press, 2005).

Risks Entailed When Setting Breakpoints

12/5/2018 • 2 minutes to read • Edit Online

When you are setting a breakpoints by specifying a memory address or a symbol plus an offset, you must not put this breakpoint in the middle of an instruction.

For example, consider the following disassembled code.

```
770000f1 5e              pop     esi
770000f2 5b              pop     ebx
770000f3 c9              leave
770000f4 c21000          ret     0x10
770000f7 837ddc00        cmp     dword ptr [ebp-0x24],0x0
```

The first three instructions are only one byte long. However, the fourth instruction is three bytes long. (It includes bytes 0x770000F4, 0x770000F5, and 0x770000F6.) If you want to put a breakpoint on this instruction by using the **bp**, **bu**, or **ba** command, you must specify the 0x770000F4 address.

If you put a breakpoint in the 0x770000F5 address by using the **ba** command, the processor puts a breakpoint at that location. But this breakpoint would never be triggered, because the processor considers 0x770000F4 to be the actual address of the instruction.

If you put a breakpoint in the 0x770000F5 address by using the **bp** or **bu** commands, the debugger writes a breakpoint at that location. However, this breakpoint might corrupt the target because of how the debugger creates breakpoints:

1. The debugger saves the contents of 0x770000F5 and overwrites this memory with a breakpoint instruction.
2. If you try to display this memory in the debugger, the debugger does not show the breakpoint instruction that it has written. Instead, the debugger shows the memory that "should" be there. That is, the debugger shows the original memory, or any modifications to that memory that you have made since inserting the breakpoint.
3. If you use the **BC** command to remove the breakpoint, the debugger restores the original memory to its proper location.

When you put a breakpoint at 0x770000F5, the debugger saves this byte and a break instruction is written here. However, when the application runs, it reaches the 0x770000F4 address and recognizes this address as the first byte of a multibyte instruction. The processor then tries to combine 0x770000F4, 0x770000F5, and possibly some later bytes into a single instruction. This combination can create a variety of behaviors, none of which are desirable.

Therefore, when you put breakpoints by using a **bp**, **bu**, or **ba** command, make sure that you always put the breakpoints at the proper address. If you are using the WinDbg graphical interface to add breakpoints, you do not have to be concerned about this situation because the correct address is chosen automatically.

Conditional breakpoints in WinDbg and other Windows debuggers

12/5/2018 • 6 minutes to read • Edit Online

Conditional breakpoints in WinDbg and other Windows debuggers are useful when you need to break in only if a specific condition is satisfied.

A conditional breakpoint is created by combining a breakpoint command with either the **j (Execute If - Else)** command or the **.if** token, followed by the **gc (Go from Conditional Breakpoint)** command. This breakpoint causes a break to occur only if a specific condition is satisfied.

The basic syntax for a conditional breakpoint using the **j** command is as follows:

```
0:000> bp Address "j (Condition) 'OptionalCommands'; 'gc' "
```

The basic syntax for a conditional breakpoint using the **.if** token is as follows:

```
0:000> bp Address ".if (Condition) {OptionalCommands} .else {gc}"
```

Conditional breakpoints are best illustrated with an example. The following command sets a breakpoint at line 143 of the Mysource.cpp source file. When this breakpoint is hit, the variable **MyVar** is tested. If this variable is less than or equal to 20, execution continues; if it is greater than 20, execution stops.

```
0:000> bp `mysource.cpp:143` "j (poi(MyVar)>0n20) ''; 'gc' "
0:000> bp `mysource.cpp:143` ".if (poi(MyVar)>0n20) {} .else {gc}"
```

The preceding command has a fairly complicated syntax that contains the following elements:

- The **bp (Set Breakpoint)** command sets breakpoints. Although the preceding example uses the bp command, you could also use the **bu (Set Unresolved Breakpoint)** command. For more information about the differences between **bp** and **bu**, and for a basic introduction to breakpoints, see Using Breakpoints.
- Source line numbers are specified by using grave accents (`). For details, see Source Line Syntax.
- When the breakpoint is hit, the command in straight quotation marks (**"**) is executed. In this example, this command is a **j (Execute If - Else)** command or an **.if** token, which tests the expression in parentheses.
- In the source program, **MyVar** is an integer. If you are using C++ expression syntax, **MyVar** is interpreted as an integer. However, in this example (and in the default debugger configuration), MASM expression syntax is used. In a MASM expression, **MyVar** is treated as an address. Thus, you need to use the **poi** operator to dereference it. (If your variable actually is a C pointer, you will need to dereference it twice--for example, **poi(poi(MyPtr))**.) The **0n** prefix specifies that this number is decimal. For syntax details, see MASM Numbers and Operators.
- The expression in parentheses is followed by two commands, surrounded by single quotation marks (**'**) for the **j** command and curly brackets ({}) for the **.if** token. If the expression is true, the first of these commands is executed. In this example, there is no first command, so command execution will end and

control will remain with the debugger. If the expression in parentheses is false, the second command will execute. The second command should almost always be a **gc (Go from Conditional Breakpoint)** command, because this command causes execution to resume in the same manner that was occurring before the breakpoint was hit (stepping, tracing, or free execution).

If you want to see a message each time the breakpoint is passed or when it is finally hit, you can use additional commands in the single quotation marks or curly brackets. For example:

```
0:000> bp `:143` "j (poi(MyVar)>5) '.echo MyVar Too Big'; '.echo MyVar Acceptable; gc' "
0:000> bp `:143` ".if (poi(MyVar)>5) {.echo MyVar Too Big} .else {.echo MyVar Acceptable; gc} "
```

These comments are especially useful if you have several such breakpoints running at the same time, because the debugger does not display its standard "Breakpoint *n* Hit" messages when you are using a command string in the **bp** command.

Conditional Breakpoint Based on String Comparison

In some situations you might want to break into the debugger only if a string variable matches a pattern. For example, suppose you want to break at kernel32!CreateEventW only if the *lpName* argument points to a string that matches the pattern "Global*". The following example shows how to create the conditional breakpoint.

```
bp kernel32!CreateEventW "$$<c:\\commands.txt"
```

The preceding **bp** command creates a breakpoint based on conditions and optional commands that are in a script file named commands.txt. The script file contains the following statements.

```
.if (@r9 != 0) { as /mu ${/v:EventName} @r9 } .else { ad /q ${/v:EventName} }
.if ($spat(@"${EventName}", "Global*") == 0)  { gc } .else { .echo EventName }
```

The *lpName* argument passed to the **CreateEventW** function is the fourth argument, so it is stored in the r9 register (x64 processor). The script performs the following steps:

1. If *lpName* is not NULL, use **as** and **${}** to create an alias named EventName. Assign to EventName the null-terminated Unicode string beginning at the address pointed to by *lpName*. On the other hand, if *lpName* is NULL, use **ad** to delete any existing alias named EventName.
2. Use **$spat** to compare the string represented by EventName to the pattern "Global*". If the string does not match the pattern, use **gc** to continue without breaking. If the string does match the pattern, break and display the string represented by EventName.

Note **$spat** performs a case-insensitive match.

Note The ampersand (@) character in $spat(@"${EventName}" specifies that the string represented by EventName is to be interpreted literally; that is, a backslash (\ ) is treated as a backslash rather than an escape character.

Conditional Breakpoints and Register Sign Extension

You can set a breakpoint that is conditional on a register value.

The following command will break at the beginning of the **myFunction** function if the **eax** register is equal to 0xA3:

```
0:000> bp mydriver!myFunction "j @eax = 0xa3  '';'gc'"
0:000> bp mydriver!myFunction ".if @eax = 0xa3  {} .else {gc}"
```

However, the following similar command will not necessarily break when **eax** equals 0xC0004321:

```
0:000> bp mydriver!myFunction "j @eax = 0xc0004321  '';'gc'"
0:000> bp mydriver!myFunction ".if @eax = 0xc0004321  {} .else {gc}"
```

The reason the preceding command will fail is that the MASM expression evaluator sign-extends registers whose high bit equals one. When **eax** has the value 0xC0004321, it will be treated as 0xFFFFFFFF`C0004321 in computations--even though **eax** will still be displayed as 0xC0004321. However, the numeral **0xc0004321** is sign-extended in kernel mode, but not in user mode. Therefore, the preceding command will not work properly in user mode. If you mask the high bits of **eax**, the command will work properly in kernel mode--but now it will fail in user mode.

You should formulate your commands defensively against sign extension in both modes. In the preceding command, you can make the command defensive by masking the high bits of a 32-bit register by using an AND operation to combine it with 0x00000000`FFFFFFFF and by masking the high bits of a numeric constant by including a grave accent ( `) in its syntax.

The following command will work properly in user mode and kernel mode:

```
0:000> bp mydriver!myFunction "j (@eax & 0x0`ffffffff) = 0x0`c0004321  '';'gc'"
0:000> bp mydriver!myFunction ".if (@eax & 0x0`ffffffff) = 0x0`c0004321  {} .else {gc}"
```

For more information about which numbers are sign-extended by the debugger, see Sign Extension.

Conditional Breakpoints in WinDbg

In WinDbg, you can create a conditional breakpoint by clicking Breakpoints from the **Edit** menu, entering a new breakpoint address into the **Command** box, and entering a condition into the **Condition** box.

For example, typing **mymod!myFunc+0x3A** into the **Command** box and **myVar < 7** into the **Condition** box is equivalent to issuing the following command:

```
0:000> bu mymod!myFunc+0x3A "j(myVar<7) '.echo "Breakpoint hit, condition myVar<7"'; 'gc'"
0:000> bu mymod!myFunc+0x3A ".if(myVar<7) {.echo "Breakpoint hit, condition myVar<7"} .else {gc}"
```

Restrictions on Conditional Breakpoints

If you are controlling the user-mode debugger from the kernel debugger, you cannot use conditional breakpoints or any other breakpoint command string that contains the **gc (Go from Conditional Breakpoint)** or **g (Go)** commands. If you use these commands, the serial interface might not be able to keep up with the number of breakpoint passes, and you will be unable to break back into CDB.

Executing Until a Specified State is Reached

12/5/2018 • 2 minutes to read • Edit Online

There are several ways to cause the target to execute until a specified state is reached.

Using a Breakpoint to Control Execution

One method is to use a breakpoint. The simplest breakpoint halts execution when the program counter reaches a specified address. A more complex breakpoint can:

- be triggered only when this address is executed by a specific thread,
- allow a specified number of passes through this address before being triggered,
- automatically issue a specified command when it is triggered, or
- watch a specified address in non-executable memory, being triggered when that memory is read or written to.

For details on how to set and control breakpoints, see Using Breakpoints.

A more complicated way to execute until a specified state is reached is to use a *conditional breakpoint*. This kind of breakpoint is set at a certain address, but is only triggered if a specified condition holds. For details, see Setting a Conditional Breakpoint.

Breakpoints and Pseudo-Registers

In specifying the desired state, it is often helpful to use *automatic pseudo-registers*. These are variables controlled by the debugger which allow you to reference a variety of values related to the target state.

For example, the following breakpoint uses the **$thread** pseudo-register, which is always equal to the value of the current thread. It resolves to the value of the current thread when it is used in a command. By using **$thread** as the argument of the **/t** parameter of the **bp (Set Breakpoint)** command, you can create a breakpoint that will be triggered every time that **NtOpenFile** is called by the thread which was active at the time you issued the **bp** command:

```
kd> bp /t @$thread nt!ntopenfile
```

This breakpoint will not be triggered when any other thread calls **NtOpenFile**.

For a list of automatic pseudo-registers, see Pseudo-Register Syntax.

Using a Script File to Control Execution

Another way to execute until a specified state is reached is to create a script file that calls itself recursively, testing the desired state in each iteration.

Typically, this script file will contain the **.if** and **.else** tokens. You can use a command such as **t (Trace)** to execute a single step, and then test the condition in question.

For example, if you wish to execute until the **eax** register contains the value 0x1234, you can create a script file called *eaxstep* that contains the following line:

```
.if (@eax == 1234) { .echo 1234 } .else { t "$<eaxstep" }
```

Then issue the following command from the Debugger Command window:

```
t "$<eaxstep"
```

This **t** command will execute a single step, and then execute the quoted command. This command happens to be **$< (Run Script File)**, which runs the *eaxstep* file. The script file tests the value of **eax**, runs the **t** command, and then calls itself recursively. This continues until the **eax** register equals 0x1234, at which point the **.echo (Echo Comment)** command prints a message to the Debugger Command window, and execution stops.

For details on script files, see Using Script Files and Using Debugger Command Programs.

Reading and Writing Memory

12/5/2018 • 2 minutes to read • Edit Online

The Windows debuggers can read and write directly into memory. This memory can be referenced by addresses or by the names of variables.

This section includes the following topics:

Accessing Memory by Virtual Address

Accessing Memory by Physical Address

Accessing Global Variables

Accessing Local Variables

Controlling Variables Through the Watch Window

Accessing Memory by Virtual Address

12/5/2018 • 2 minutes to read • Edit Online

To access memory addresses or address ranges, you can use several commands. Visual Studio and WinDbg provide user interface elements (as well as commands) that you can use to view and edit memory. For more information, see Viewing and Editing Memory and Registers in Visual Studio and Viewing and Editing Memory in WinDbg.

The following commands can read or write memory in a variety of formats. These formats include hexadecimal bytes, words (words, double words, and quad-words), integers (short, long, and quad integers and unsigned integers), floating-point numbers (10-byte, 16-byte, 32-byte, and 64-byte real numbers), and ASCII characters.

- The **d* (Display Memory)** command displays the contents of a specified memory address or range.
- The **e* (Enter Values)** command writes a value to the specified memory address.

You can use the following commands to handle more specialized data types:

- The **dt (Display Type)** command finds a variety of data types and displays data structures that have been created by the application that is being debugged. This command is highly versatile and has many variations and options.
- The **ds, dS (Display String)** command displays a STRING, ANSI_STRING, or UNICODE_STRING data structure.
- The **dl (Display Linked List)** command traces and displays a linked list.
- The **d*s (Display Words and Symbols)** command finds double-words or quad-words that might contain symbol information and then displays the data and the symbol information.
- The **!address** extension command displays information about the properties of the memory that is located at a specific address.

You can use the following commands to manipulate memory ranges:

- The **m (Move Memory)** command moves the contents of one memory range to another.
- The **f (Fill Memory)** command writes a pattern to a memory range, repeating it until the range is full.
- The **c (Compare Memory)** command compares the contents of two memory ranges.
- The **s (Search Memory)** command searches for a specified pattern within a memory range or searches for any ASCII or Unicode characters that exist in a memory range.
- The **.holdmem (Hold and Compare Memory)** command compares one memory range to another.

In most situations, these commands interpret their parameters in the current radix. Therefore, you should add **0x** before hexadecimal addresses if the current radix is not 16. However, the display output of these commands is typically in hexadecimal format, regardless of the current radix. (For more information about the output, see the individual command topics.) The Memory window displays integers and real numbers in decimal format and displays other formats in hexadecimal format.

To change the default radix, use the **n (Set Number Base)** command. To quickly convert numbers from one base to another, use the **? (Evaluate Expression)** command or the **.formats (Show Number Formats)** command.

When you are performing user-mode debugging, the meaning of virtual addresses is determined by the current process. When you are performing kernel-mode debugging, the meaning of virtual addresses can be controlled by the debugger. For more information, see Process Context.

Accessing Memory by Physical Address

12/5/2018 • 2 minutes to read • Edit Online

To read from a physical address, use the **!db**, **!dc**, **!dd**, **!dp**, **!du**, and **!dw** extension commands.

To write to a physical address, use the **!eb** and **!ed** extension commands.

The **fp (Fill Physical Memory)** command writes a pattern to a physical memory range, repeating it until the range is full.

When you are using WinDbg in kernel mode, you can also read or write to physical memory directly from the Memory window.

To search physical memory for a piece of data or a range of data, use the **!search** extension command.

Also, for more information about physical addresses, see Converting Virtual Addresses to Physical Addresses.

Accessing Global Variables

12/5/2018 • 2 minutes to read • Edit Online

The names of global variables are stored in the symbol files that are created when an application is compiled. The debugger interprets the name of a global variable as a virtual address. Any command that accepts an address as a parameter also accepts the name of a variable. Therefore, you can use all of the commands that are described in Accessing Memory by Virtual Address to read or write global variables.

In addition, you can use the **? (Evaluate Expression)** command to display the address that is associated with any symbol.

Visual Studio and WinDbg provide user interface elements that you can use (in addition to commands) to view and edit global variables. See Viewing and Editing Memory and Registers in Visual Studio and Viewing and Editing Global Variables in WinDbg.

Consider the following example. Suppose that you want to examine the `MyCounter` global variable, which is a 32-bit integer. Also suppose that the default radix is 10.

You can obtain this variable's address and then display it as follows.

```
0:000> ? MyCounter
Evaluate expression: 1244892 = 0012fedc
0:000> dd 0x0012fedc L1
0012fedc  00000052
```

The first command output tells you that the address of `MyCounter` is 0x0012FEDC. You can then use the **d* (Display Memory)** command to display one double-word at this address. (You could also use 1244892, which is the decimal version of this address. However, most C programmers prefer to use 0x0012FEDC.) The second command tells you that the value of MyCounter is 0x52 (decimal 82).

You could also perform these steps in the following command.

```
0:000> dd MyCounter L1
0012fedc  00000052
```

To change the value of `MyCounter` to decimal 83, use the following command.

```
0:000> ed MyCounter 83
```

This example uses decimal input, because that format seems more natural for an integer. However, the output of the **d*** command is still in hexadecimal format.

```
0:000> dd MyCounter L1 0012fedc  00000053
```

Accessing Local Variables

12/5/2018 • 2 minutes to read • Edit Online

Local variables, like global variables, are stored in the symbol files. And as with global variables, the debugger interprets their names as addresses. They can be read and written in the same manner as global variables. However, if you need to indicate to a command that a symbol is local, precede the symbol with a dollar sign ($) and an exclamation point (!), as in `$!var`.

Visual Studio and WinDbg provide user interface elements that you can use (in addition to commands) to view and edit local variables. For more information, see Viewing and Editing Memory and Registers in Visual Studio and Viewing and Editing Local Variables in WinDbg.

You can also use the following methods to display, change, and use local variables:

- The **dv (Display Local Variables)** command displays the names and values of all local variables.
- The **!for_each_local** extension enables you to execute a single command repeatedly, once for each local variable.

However, there is one primary difference between local and global variables. When an application is executing, the meaning of local variables depends on the location of the program counter, because the scope of such variables extends only to the function in which they are defined.

The debugger interprets local variables according to the local context. By default, this context matches the location of the program counter. But the debugger can change the context. For more information about the local context, see Local Context.

When the local context is changed, the Locals window is immediately updated to reflect the new collection of local variables. The **dv** command also shows the new variables. All of these variable names are then interpreted correctly by the memory commands that are described earlier. You can then read or write to these variables.

Controlling Variables Through the Watch Window

12/5/2018 • 2 minutes to read • Edit Online

In WinDbg, you can also use the Watch window to display and change global and local variables.

The Watch window can display any list of variables that you want. These variables can include global variables and local variables from any function. At any time, the Watch window displays the values of those variables that match the current function's scope. You can also change the values of these variables through the Watch window.

Unlike the Locals window, the Watch window is not affected by changes to the local context. Only those variables that are defined in the scope of the current program counter can have their values displayed or modified.

Converting Virtual Addresses to Physical Addresses

12/5/2018 • 5 minutes to read • Edit Online

Most debugger commands use virtual addresses, not physical addresses, as their input and output. However, there are times that having the physical address can be useful.

There are two ways to convert a virtual address to a physical address: by using the **!vtop** extension, and by using the **!pte** extension.

For an overview of virtual address in Windows, see Virtual address spaces.

Address Conversion Using !vtop

Suppose you are debugging a target computer on which the MyApp.exe process is running and you want to investigate the virtual address 0x0012F980. Here is the procedure you would use with the **!vtop** extension to determine the corresponding physical address.

Converting a virtual address to a physical address using !vtop

1. Make sure that you are working in hexadecimal. If necessary, set the current base with the **N 16** command.
2. Determine the *byte index* of the address. This number is equal to the lowest 12 bits of the virtual address. Thus, the virtual address 0x0012F980 has a byte index of 0x980.
3. Determine the *directory base* of the address by using the **!process** extension:

```
kd> !process 0 0
**** NT ACTIVE PROCESS DUMP ****
....
PROCESS ff779190  SessionId: 0  Cid: 04fc    Peb: 7ffdf000  ParentCid: 0394
 DirBase: 098fd000  ObjectTable: e1646b30  TableSize:   8.
    Image: MyApp.exe
```

4. Determine the *page frame number* of the directory base. This is simply the directory base without the three trailing hexadecimal zeros. In this example, the directory base is 0x098FD000, so the page frame number is 0x098FD.
5. Use the **!vtop** extension. The first parameter of this extension should be the page frame number. The second parameter of **!vtop** should be the virtual address in question:

```
kd> !vtop 98fd 12f980
Pdi 0 Pti 12f
0012f980 09de9000 pfn(09de9)
```

   The second number shown on the final line is the physical address of the beginning of the physical page.
6. Add the byte index to the address of the beginning of the page: 0x09DE9000 + 0x980 = 0x09DE9980. This is the desired physical address.

You can verify that this computation was done correctly by displaying the memory at each address. The **!d*** extension displays memory at a specified physical address:

```
kd> !dc 9de9980
# 9de9980 6d206e49 726f6d65 00120079 0012f9f4 In memory.......
# 9de9990 0012f9f8 77e57119 77e8e618 ffffffff .....q.w...w....
# 9de99a0 77e727e0 77f6f13e 77f747e0 ffffffff .'.w>..w.G.w....
# 9de99b0 .....
```

The **d* (Display Memory)** command uses a virtual address as its argument:

```
kd> dc 12f980
0012f980  6d206e49 726f6d65 00120079 0012f9f4  In memory.......
0012f990  0012f9f8 77e57119 77e8e618 ffffffff  .....q.w...w....
0012f9a0  77e727e0 77f6f13e 77f747e0 ffffffff  .'.w>..w.G.w....
0012f9b0  .....
```

Because the results are the same, this indicates that the physical address 0x09DE9980 does indeed correspond to the virtual address 0x0012F980.

Address Conversion Using !pte

Again, assume you are investigating the virtual address 0x0012F980 belonging to the MyApp.exe process. Here is the procedure you would use with the **!pte** extension to determine the corresponding physical address:

Converting a virtual address to a physical address using !pte

1. Make sure that you are working in hexadecimal. If necessary, set the current base with the **N 16** command.
2. Determine the *byte index* of the address. This number is equal to the lowest 12 bits of the virtual address. Thus, the virtual address 0x0012F980 has a byte index of 0x980.
3. Set the process context to the desired process:

   ```
   kd> !process 0 0
   **** NT ACTIVE PROCESS DUMP ****
   ....
   PROCESS ff779190  SessionId: 0  Cid: 04fc    Peb: 7ffdf000  ParentCid: 0394
       DirBase: 098fd000  ObjectTable: e1646b30  TableSize:   8.
       Image: MyApp.exe

   kd> .process /p ff779190
   Implicit process is now ff779190
   .cache forcedecodeuser done
   ```

4. Use the **!pte** extension with the virtual address as its argument. This displays information in two columns. The left column describes the page directory entry (PDE) for this address; the right column describes its page table entry (PTE):

   ```
   kd> !pte 12f980
                  VA 0012f980
   PDE at   C0300000        PTE at C00004BC
   contains 0BA58067      contains 09DE9067
   pfn ba58 ---DA--UWV    pfn 9de9 ---DA--UWV
   ```

5. Look in the last row of the right column. The notation "pfn 9de9" appears. The number 0x9DE9 is the *page frame number* (PFN) of this PTE. Multiply the page frame number by 0x1000 (for example, shift it left 12 bits). The result, 0x09DE9000, is the physical address of the beginning of the page.
6. Add the byte index to the address of the beginning of the page: 0x09DE9000 + 0x980 = 0x09DE9980. This is the desired physical address.

This is the same result obtained by the earlier method.

Converting Addresses By Hand

Although the **!ptov** and **pte** extensions supply the fastest way to convert virtual addresses to physical addresses, this conversion can be done manually as well. A description of this process will shed light on some of the details of the virtual memory architecture.

Memory structures vary in size, depending on the processor and the hardware configuration. This example is taken from an x86 system that does not have Physical Address Extension (PAE) enabled.

Using 0x0012F980 again as the virtual address, you first need to convert it to binary, either by hand or by using the **.formats (Show Number Formats)** command:

```
kd> .formats 12f980
Evaluate expression:
  Hex:     0012f980
  Decimal: 1243520
  Octal:   00004574600
  Binary:  00000000 00010010 11111001 10000000
  Chars:   ....
  Time:    Thu Jan 15 01:25:20 1970
  Float:   low 1.74254e-039 high 0
  Double:  6.14381e-318
```

This virtual address is a combination of three fields. Bits 0 to 11 are the byte index. Bits 12 to 21 are the page table index. Bits 22 to 31 are the page directory index. Separating the fields, you have:

```
0x0012F980  =  0y  00000000 00   010010 1111   1001 10000000
```

This exposes the three parts of the virtual address:

- Page directory index = 0y0000000000 = 0x0
- Page table index = 0y0100101111 = 0x12F
- Byte index = 0y100110000000 = 0x980

You then need three additional pieces of information for your system.

- The size of each PTE. This is 4 bytes on non-PAE x86 systems.
- The size of a page. This is 0x1000 bytes.
- The PTE_BASE virtual address. On a non-PAE system, this is 0xC0000000.

Using this data, you can compute the address of the PTE itself:

```
PTE address   =   PTE_BASE
                + (page directory index) * PAGE_SIZE
                + (page table index) * sizeof(MMPTE)
              =   0xc0000000
                + 0x0   * 0x1000
                + 0x12F * 4
              =   0xC00004BC
```

This is the address of the PTE. The PTE is a 32-bit DWORD. Examine its contents:

```
kd> dd 0xc00004bc L1
c00004bc  09de9067
```

This PTE has value 0x09DE9067. It is made of two fields:

- The low 12 bits of the PTE are the *status flags*. In this case, these flags equal 0x067 -- or in binary, 0y000001100111. For an explanation of the status flags, see the **!pte** reference page.
- The high 20 bits of the PTE are equal to the *page frame number* (PFN) of the PTE. In this case, the PFN is 0x09DE9.

The first physical address on the physical page is the PFN multiplied by 0x1000 (shifted left 12 bits). The byte index is the offset on this page. Thus,the physical address you are looking for is 0x09DE9000 + 0x980 = 0x09DE9980. This is the same result obtained by the earlier methods.

Using the !analyze Extension

4/12/2019 • 10 minutes to read • Edit Online

The first step in debugging a crashed target computer or application is to use the **!analyze** extension command.

This extension performs a tremendous amount of automated analysis. The results of this analysis are displayed in the Debugger Command window.

You should use the **-v** option for a fully verbose display of data. For details on other options, see the **!analyze** reference page.

This topic contains:

- A User-Mode !analyze -v Example
- A Kernel-Mode !analyze -v Example
- The Followup Field and the triage.ini File
- Additional !analyze Techniques

A User-Mode !analyze -v Example

In this example, the debugger is attached to a user-mode application that has encountered an exception.

```
0:000> !analyze -v
*******************************************************************************
*                                                                             *
*                        Exception Analysis                                   *
*                                                                             *
*******************************************************************************

Debugger SolutionDb Connection::Open failed 80004005
```

If you are connected to the internet, the debugger attempts to access a database of crash solutions maintained by Microsoft. In this case, an error message was displayed, indicating that either your machine was unable to access the internet or the web site was not working.

```
FAULTING_IP:
ntdll!PropertyLengthAsVariant+73
77f97704 cc               int     3
```

The FAULTING_IP field shows the instruction pointer at the time of the fault.

```
EXCEPTION_RECORD:  ffffffff -- (.exr ffffffffffffffff)
ExceptionAddress: 77f97704 (ntdll!PropertyLengthAsVariant+0x00000073)
   ExceptionCode: 80000003 (Break instruction exception)
  ExceptionFlags: 00000000
NumberParameters: 3
   Parameter[0]: 00000000
   Parameter[1]: 00010101
   Parameter[2]: ffffffff
```

The EXCEPTION_RECORD field shows the exception record for this crash. This information can also be viewed by using the **.exr (Display Exception Record)** command.

```
BUGCHECK_STR:  80000003
```

The BUGCHECK_STR field shows the exception code. The name is a misnomer—the term *bug check* actually signifies a kernel-mode crash. In user-mode debugging, the exception code will be displayed—in this case, 0x80000003.

```
DEFAULT_BUCKET_ID:  APPLICATION_FAULT
```

The DEFAULT_BUCKET_ID field shows the general category of failures that this failure belongs to.

```
PROCESS_NAME:  MyApp.exe
```

The PROCESS_NAME field specifies the name of the process that raised the exception.

```
LAST_CONTROL_TRANSFER:  from 01050963 to 77f97704
```

The LAST_CONTROL_TRANSFER field shows the last call on the stack. In this case, the code at address 0x01050963 called a function at 0x77F97704. You can use these addresses with the **ln (List Nearest Symbols)** command to determine what modules and functions these addresses reside in.

```
STACK_TEXT:
0006b9dc 01050963 00000000 0006ba04 000603fd ntdll!PropertyLengthAsVariant+0x73
0006b9f0 010509af 00000002 0006ba04 77e1a449 MyApp!FatalErrorBox+0x55 [D:\source_files\MyApp\util.c @ 541]
0006da04 01029f4e 01069850 0000034f 01069828 MyApp!ShowAssert+0x47 [D:\source_files\MyApp\util.c @ 579]
0006db6c 010590c3 000e01ea 0006fee4 0006feec MyApp!SelectColor+0x103 [D:\source_files\MyApp\colors.c @ 849]
0006fe04 77e11d0a 000e01ea 00000111 0000413c MyApp!MainWndProc+0x1322 [D:\source_files\MyApp\MyApp.c @ 1031]
0006fe24 77e11bc8 01057da1 000e01ea 00000111 USER32!UserCallWinProc+0x18
0006feb0 77e172b4 0006fee4 00000001 010518bf USER32!DispatchMessageWorker+0x2d0
0006febc 010518bf 0006fee4 00000000 01057c5d USER32!DispatchMessageA+0xb
0006fec8 01057c5d 0006fee4 77f82b95 77f83920 MyApp!ProcessQCQPMessage+0x3b [D:\source_files\MyApp\util.c @
2212]
0006ff70 01062cbf 00000001 00683ed8 00682b88 MyApp!main+0x1e6 [D:\source_files\MyApp\MyApp.c @ 263]
0006ffc0 77e9ca90 77f82b95 77f83920 7ffdf000 MyApp!mainCRTStartup+0xff [D:\source_files\MyApp\crtexe.c @ 338]
0006fff0 00000000 01062bc0 00000000 000000c8 KERNEL32!BaseProcessStart+0x3d
```

The STACK_TEXT field shows a stack trace of the faulting component.

```
FOLLOWUP_IP:
MyApp!FatalErrorBox+55
01050963 5e              pop     esi

FOLLOWUP_NAME:  dbg

SYMBOL_NAME:  MyApp!FatalErrorBox+55

MODULE_NAME:  MyApp

IMAGE_NAME:  MyApp.exe

DEBUG_FLR_IMAGE_TIMESTAMP:  383490a9
```

When **!analyze** determines the instruction that has probably caused the error, it displays it in the FOLLOWUP_IP field. The SYMBOL_NAME, MODULE_NAME, IMAGE_NAME, and DEBUG_FLR_IMAGE_TIMESTAMP fields show the symbol, module, image name, and image timestamp corresponding to this instruction.

```
STACK_COMMAND:  .ecxr ; kb
```

The STACK_COMMAND field shows the command that was used to obtain the STACK_TEXT. You can use this command to repeat this stack trace display, or alter it to obtain related stack information.

```
BUCKET_ID:  80000003_MyApp!FatalErrorBox+55
```

The BUCKET_ID field shows the specific category of failures that the current failure belongs to. This category helps the debugger determine what other information to display in the analysis output.

```
Followup: dbg
---------
```

For information about the FOLLOWUP_NAME and the Followup fields, see The Followup Field and the triage.ini File.

There are a variety of other fields that may appear:

- If control was transferred to an invalid address, then the FAULTING_IP field will contain this invalid address. Instead of the FOLLOWUP_IP field, the FAILED_INSTRUCTION_ADDRESS field will show the disassembled code from this address, although this disassembly will probably be meaningless. In this situation, the SYMBOL_NAME, MODULE_NAME, IMAGE_NAME, and DEBUG_FLR_IMAGE_TIMESTAMP fields will refer to the caller of this instruction.
- If the processor misfires, you may see the SINGLE_BIT_ERROR, TWO_BIT_ERROR, or POSSIBLE_INVALID_CONTROL_TRANSFER fields.
- If memory corruption seems to have occurred, the CHKIMG_EXTENSION field will specify the **!chkimg** extension command that should be used to investigate.

A Kernel-Mode !analyze -v Example

In this example, the debugger is attached to a computer that has just crashed.

```
kd> !analyze -v
*******************************************************************************
*                                                                             *
*                        Bugcheck Analysis                                    *
*                                                                             *
*******************************************************************************

DRIVER_IRQL_NOT_LESS_OR_EQUAL (d1)
An attempt was made to access a pagable (or completely invalid) address at an
interrupt request level (IRQL) that is too high.  This is usually
caused by drivers using improper addresses.
If kernel debugger is available get stack backtrace.
```

The first element of the display shows the bug check code and information about this type of bug check. Some of the text displayed may not apply to this specific instance. For more details on each bug check, see the Bug Check Code Reference section.

```
Arguments:
Arg1: 00000004, memory referenced
Arg2: 00000002, IRQL
Arg3: 00000001, value 0 = read operation, 1 = write operation
Arg4: f832035c, address which referenced memory
```

The bug check parameters are displayed next. They are each followed by a description. For example, the third parameter is 1, and the comment following it explains that this indicates that a write operation failed.

```
## Debugging Details:


WRITE_ADDRESS:  00000004 Nonpaged pool

CURRENT_IRQL:  2
```

The next few fields vary depending on the nature of the crash. In this case, we see WRITE_ADDRESS and CURRENT_IRQL fields. These are simply restating the information shown in the bug check parameters. By comparing the statement "Nonpaged pool" to the bug check text that reads "an attempt was made to access a pagable (or completely invalid) address," we can see that the address was invalid. The invalid address in this case was 0x00000004.

```
FAULTING_IP:
USBPORT!USBPORT_BadRequestFlush+7c
f832035c 894204          mov     [edx+0x4],eax
```

The FAULTING_IP field shows the instruction pointer at the time of the fault.

```
DEFAULT_BUCKET_ID:  DRIVER_FAULT
```

The DEFAULT_BUCKET_ID field shows the general category of failures that this failure belongs to.

```
BUGCHECK_STR:  0xD1
```

The BUGCHECK_STR field shows the bug check code, which we have already seen. In some cases additional triage information is appended.

```
TRAP_FRAME:  f8950dfc -- (.trap ffffffff8950dfc)
.trap ffffffff8950dfc
ErrCode = 00000002
eax=81cc86dc ebx=81cc80e0 ecx=81e55688 edx=00000000 esi=81cc8028 edi=8052cf3c
eip=f832035c esp=f8950e70 ebp=f8950e90 iopl=0         nv up ei pl nz ac po nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00010216
USBPORT!USBPORT_BadRequestFlush+7c:
f832035c 894204          mov     [edx+0x4],eax     ds:0023:00000004=????????
.trap
Resetting default context
```

The TRAP_FRAME field shows the trap frame for this crash. This information can also be viewed by using the **.trap (Display Trap Frame)** command.

```
LAST_CONTROL_TRANSFER:  from f83206e0 to f832035c
```

The LAST_CONTROL_TRANSFER field shows the last call on the stack. In this case, the code at address 0xF83206E0 called a function at 0xF832035C. You can use the **ln (List Nearest Symbols)** command to determine what module and function these addresses reside in.

```
STACK_TEXT:
f8950e90 f83206e0 024c7262 00000000 f8950edc USBPORT!USBPORT_BadRequestFlush+0x7c
f8950eb0 804f5561 81cc8644 81cc8028 6d9a2f30 USBPORT!USBPORT_DM_TimerDpc+0x10c
f8950fb4 804f5644 6e4be98e 00000000 ffdff000 nt!KiTimerListExpire+0xf3
f8950fe0 8052c47c 8053cf20 00000000 00002e42 nt!KiTimerExpiration+0xb0
f8950ff4 8052c16a efdefd44 00000000 00000000 nt!KiRetireDpcList+0x31
```

The STACK_TEXT field shows a stack trace of the faulting component.

```
FOLLOWUP_IP:
USBPORT!USBPORT_BadRequestFlush+7c
f832035c 894204          mov     [edx+0x4],eax
```

The FOLLOWUP_IP field shows the disassembly of the instruction that has probably caused the error.

```
FOLLOWUP_NAME:  usbtri

SYMBOL_NAME:  USBPORT!USBPORT_BadRequestFlush+7c

MODULE_NAME:  USBPORT

IMAGE_NAME:  USBPORT.SYS

DEBUG_FLR_IMAGE_TIMESTAMP:  3b7d868b
```

The SYMBOL_NAME, MODULE_NAME, IMAGE_NAME, and DBG_FLR_IMAGE_TIMESTAMP fields show the symbol, module, image, and image timestamp corresponding to this instruction (if it is valid), or to the caller of this instruction (if it is not).

```
STACK_COMMAND:  .trap ffffffff8950dfc ; kb
```

The STACK_COMMAND field shows the command that was used to obtain the STACK_TEXT. You can use this command to repeat this stack trace display, or alter it to obtain related stack information.

```
BUCKET_ID:  0xD1_W_USBPORT!USBPORT_BadRequestFlush+7c
```

The BUCKET_ID field shows the specific category of failures that the current failure belongs to. This category helps the debugger determine what other information to display in the analysis output.

```
INTERNAL_SOLUTION_TEXT:  https://oca.microsoft.com/resredir.asp?sid=62&State=1
```

If you are connected to the internet, the debugger attempts to access a database of crash solutions maintained by Microsoft. This database contains links to a tremendous number of Web pages that have information about known bugs. If a match is found for your problem, the INTERNAL_SOLUTION_TEXT field will show a URL that you can access for more information.

```
Followup: usbtri
---------

      This problem has a known fix.
      Please connect to the following URL for details:
      -----------------------------------------------
      https://oca.microsoft.com/resredir.asp?sid=62&State=1
```

For information about the FOLLOWUP_NAME and the Followup fields, see The Followup Field and the triage.ini File:

There are a variety of other fields that may appear:

- If control was transferred to an invalid address, then the FAULTING_IP field will contain this invalid address. Instead of the FOLLOWUP_IP field, the FAILED_INSTRUCTION_ADDRESS field will show the disassembled code from this address, although this disassembly will probably be meaningless. In this situation, the SYMBOL_NAME, MODULE_NAME, IMAGE_NAME, and DBG_FLR_IMAGE_TIMESTAMP fields will refer to the caller of this instruction.
- If the processor misfires, you may see the SINGLE_BIT_ERROR, TWO_BIT_ERROR, or POSSIBLE_INVALID_CONTROL_TRANSFER fields.
- If memory corruption seems to have occurred, the CHKIMG_EXTENSION field will specify the **!chkimg** extension command that should be used to investigate.
- If a bug check occurred within the code of a device driver, its name may be displayed in the BUGCHECKING_DRIVER field.

The Followup Field and the triage.ini File

In both user mode and kernel mode, the Followup field in the display will show information about the owner of the current stack frame, if this can be determined. This information is determined in the following manner:

1. When the **!analyze** extension is used, the debugger begins with the top frame in the stack and determines whether it is responsible for the error. If it isn't, the next frame is analyzed. This process continues until a frame that might be at fault is found.
2. The debugger attempts to determine the owner of the module and function in this frame. If the owner can be determined, this frame is considered to be at fault.
3. If the owner cannot be determined, the debugger passes to the next stack frame, and so on, until the owner is determined (or the stack is completely examined). The first frame whose owner is found in this search is considered to be at fault. If the stack is exhausted without any information being found, no Followup field is displayed.
4. The owner of the frame at fault is displayed in the Followup field. If **!analyze -v** is used, the FOLLOWUP_IP, SYMBOL_NAME, MODULE_NAME, IMAGE_NAME, and DBG_FLR_IMAGE_TIMESTAMP fields will refer to this frame.

For the Followup field to display useful information, you must first create a Triage.ini file containing the names of the module and function owners.

The Triage.ini file should identify the owners of all modules that could possibly have errors. You can use an informational string instead of an actual owner, but this string cannot contain spaces. If you are certain that a module will not fault, you can omit this module or indicate that it should be skipped. It is also possible to specify owners of individual functions, giving the triage process an even finer granularity.

For details on the syntax of the Triage.ini file, see Specifying Module and Function Owners.

Additional !analyze Techniques

If you do not believe that the BUCKET_ID is correct, you can override the bucket choice by using **!analyze** with the **-D** parameter.

If no crash or exception has occurred, **!analyze** will display a very short text giving the current status of the target. In certain situations you may want to force the analysis to take place as if a crash had occurred. Use **!analyze -f** to accomplish this task.

In user mode, if an exception has occurred but you believe the underlying problem is a hung thread, set the current thread to the thread you are investigating, and then use **!analyze -hang**. This extension will perform a thread stack analysis to determine if any threads are blocking other threads.

In kernel mode, if a bug check has occurred but you believe the underlying problem is a hung thread, use **!analyze -hang**. This extension will investigate locks held by the system and scan the DPC queue chain, and will display any indications of hung threads. If you believe the problem is a kernel-mode resource deadlock, use the **!deadlock** extension along with the **Deadlock Detection** option of Driver Verifier.

You can also automatically ignore known issues. To do this, you must first create an XML file containing a formatted list of known issues. Use the **!analyze -c -load***KnownIssuesFile* extension to load this file. Then when an exception or break occurs, use the **!analyze -c** extension. If the exception matches one of the known issues, the target will resume execution. If the target does not resume executing, then you can use **!analyze -v** to determine the cause of the problem. A sample XML file can be found in the sdk\samples\analyze_continue subdirectory of the debugger installation directory.

Handling a Bug Check When Driver Verifier is Enabled

12/5/2018 • 3 minutes to read • Edit Online

Driver Verifier detects driver errors at run time. You can use Driver Verifier along with the **!analyze** debugger command to detect and display information about errors in your driver.

In Windows 8, Driver Verifier has been enhanced with new features, including DDI Compliance Checking. Here we give an example that demonstrates DDI Compliance Checking.

Use the following procedure to get set up.

1. Establish a kernel-mode debugging session between a host and target computer.
2. Install your driver on the target computer.
3. On the target computer, open a Command Prompt window and enter the command **verifier**. Use Driver Verifier Manager to enable Driver Verifier for your driver.
4. Reboot the target computer.

When Driver Verifier detects an error, it generates a bug check. Then Windows breaks into the debugger and displays a brief description of the error. Here is an example where Driver Verifier generates Bug Check **DRIVER_VERIFIER_DETECTED_VIOLATION (C4)**.

```
Driver Verifier: Extension abort with Error Code 0x20005
Error String ExAcquireFastMutex should only be called at IRQL <= APC_LEVEL.

*** Fatal System Error: 0x000000c4
                       (0x0000000000020005,0xFFFFF88000E16F50,0x0000000000000000,0x0000000000000000)

Break instruction exception - code 80000003 (first chance)

A fatal system error has occurred.
Debugger entered on first try; Bugcheck callbacks have not been invoked.

A fatal system error has occurred.

nt!DbgBreakPointWithStatus:
fffff802`a40ef930 cc              int     3
```

In the debugger, enter **!analyze -v** to get a detailed description of the error.

```
0: kd> !analyze -v
Connected to Windows 8 9200 x64 target at (Thu Oct 11 13:48:31.270 2012 (UTC - 7:00)), ptr64 TRUE
Loading Kernel Symbols
...............................................................
...............................................................
..................
Loading User Symbols
...........................................
Loading unloaded module list
............
*******************************************************************************
*                                                                             *
*                        Bugcheck Analysis                                    *
*                                                                             *
*******************************************************************************

DRIVER_VERIFIER_DETECTED_VIOLATION (c4)
A device driver attempting to corrupt the system has been caught.  This is
because the driver was specified in the registry as being suspect (by the
administrator) and the kernel has enabled substantial checking of this driver.
If the driver attempts to corrupt the system, bugchecks 0xC4, 0xC1 and 0xA will
be among the most commonly seen crashes.
Arguments:
Arg1: 0000000000020005, ID of the 'IrqlExApcLte1' rule that was violated.
Arg2: fffff88000e16f50, A pointer to the string describing the violated rule condition.
Arg3: 0000000000000000, An optional pointer to the rule state variable(s).
Arg4: 0000000000000000, Reserved (unused)

## Debugging Details:

...

DV_VIOLATED_CONDITION:  ExAcquireFastMutex should only be called at IRQL <= APC_LEVEL.

DV_MSDN_LINK: !url https://go.microsoft.com/fwlink/p/?linkid=216022

DV_RULE_INFO: !ruleinfo 0x20005

BUGCHECK_STR:  0xc4_IrqlExApcLte1_XDV

DEFAULT_BUCKET_ID:  WIN8_DRIVER_FAULT

PROCESS_NAME:  TiWorker.exe

CURRENT_IRQL:  9
```

In the preceding output, you can see the name and description of the rule, **IrqlExApcLte1**, that was violated, and you can click a link to the reference page that describes the rule: https://go.microsoft.com/fwlink/p/?linkid=216022. You can also click a debugger command link, **!ruleinfo 0x20005**, to get information about the rule. In this case, the rule states that you cannot call ExAcquireFastMutex if the interrupt request level (IRQL) is greater than APC_LEVEL. The output shows that the current IRQL is 9, and in wdm.h you can see that APC_LEVEL has a value of 1. For more information about IRQLs, see Managing Hardware Priorities.

The output of **!analyze -v** continues with a stack trace and information about the code that caused the error. In the following output, you can see that the **OnInterrupt** routine in MyDriver.sys called ExAcquireFastMutex. **OnInterrupt** is an interrupt service routine that runs at an IRQL greater than APC_LEVEL, so it is a violation for this routine to call ExAcquireFastMutex.

```
LAST_CONTROL_TRANSFER:  from fffff802a41f00ea to fffff802a40ef930

STACK_TEXT:
... : nt!DbgBreakPointWithStatus ...
... : nt!KiBugCheckDebugBreak+0x12 ...
... : nt!KeBugCheck2+0x79f ...
... : nt!KeBugCheckEx+0x104 ...
... : VerifierExt!SLIC_abort+0x47 ...
... : VerifierExt!SLIC_ExAcquireFastMutex_entry_irqlexapclte1+0x25 ...
... : VerifierExt!ExAcquireFastMutex_wrapper+0x1a ...
... : nt!ViExAcquireFastMutexCommon+0x1d ...
... : nt!VerifierExAcquireFastMutex+0x1a ...
... : MyDriver!OnInterrupt+0x69 ...
... : nt!KiScanInterruptObjectList+0x6f ...
... : nt!KiChainedDispatch+0x19a ...
...

STACK_COMMAND:  kb

FOLLOWUP_IP:
MyDriver!OnInterrupt+69 ...
fffff880`16306649 4c8d0510040000  lea     r8,[MyDriver! ?? ::FNODOBFM::`string' (fffff880`16306a60)]

FAULTING_SOURCE_LINE:  c:\MyDriverwdm03\cpp\MyDriver\pnp.c

FAULTING_SOURCE_FILE:  c:\MyDriverwdm03\cpp\MyDriver\pnp.c

FAULTING_SOURCE_LINE_NUMBER:  26

FAULTING_SOURCE_CODE:
    22:    if(1 == interruptStatus)
    23:    {
    24:       ...
    25:       ExAcquireFastMutex( &(fdoExt->fastMutex) );
>   26:       ...
    27:       ExReleaseFastMutex( &(fdoExt->fastMutex) );
    28:       ...
    29:       return TRUE;
    30:    }
    31:    else


SYMBOL_STACK_INDEX:  9

SYMBOL_NAME:  MyDriver!OnInterrupt+69

FOLLOWUP_NAME:  ...

MODULE_NAME: MyDriver

IMAGE_NAME:  MyDriver.sys

DEBUG_FLR_IMAGE_TIMESTAMP:  50772f37

BUCKET_ID_FUNC_OFFSET:  69

FAILURE_BUCKET_ID:  0xc4_IrqlExApcLte1_XDV_VRF_MyDriver!OnInterrupt

BUCKET_ID:  0xc4_IrqlExApcLte1_XDV_VRF_MyDriver!OnInterrupt
```

Related topics

Static Driver Verifier

Noninvasive Debugging (User Mode)

12/5/2018 • 2 minutes to read • Edit Online

If a user-mode application is already running, the debugger can debug it *noninvasively*. With noninvasive debugging, you do not have as many debugging actions. However, you can minimize the debugger's interference with the target application. Noninvasive debugging is useful if the target application has stopped responding.

In noninvasive debugging, the debugger does not actually attach to the target application. The debugger suspends all of the target's threads and has access to the target's memory, registers, and other such information. However, the debugger cannot control the target, so commands like **g (Go)** do not work.

If you try to execute commands that are not permitted during noninvasive debugging, you receive an error message that states, "The debugger is not attached, so process execution cannot be monitored."

Selecting the Process to Debug

You can specify the target application by the process ID (PID) or process name.

If you specify the application by name, you should use the complete name of the process, including the file name extension. If two processes have the same name, you must use the process ID instead.

For more information about how to determine the process ID and the process name, see Finding the Process ID.

For information about starting and stopping a noninvasive debugging session, see the following topics:

- Debugging a User-Mode Process Using Visual Studio
- Debugging a User-Mode Process Using WinDbg
- Debugging a User-Mode Process Using CDB

CDB Command Line

To noninvasively debug a running process from the CDB command line, specify the -pv option, the -p option, and the process ID, in the following syntax.

cdb -pv -p *ProcessID*

Or, to noninvasively debug a running process by specifying the process name, use the following syntax instead.

cdb -pv -pn *ProcessName*

There are several other useful command-line options. For more information about the command-line syntax, see **CDB Command-Line Options**.

WinDbg Command Line

To noninvasively debug a running process from the WinDbg command line, specify the -pv option, the -p option, and the process ID, in the following syntax.

windbg -pv -p *ProcessID*

Or, to noninvasively debug a running process by specifying the process name, use the following syntax instead.

windbg -pv -pn *ProcessName*

There are several other useful command-line options. For more information about the command-line syntax, see **WinDbg Command-Line Options**.

WinDbg Menu

When WinDbg is in dormant mode, you can noninvasively debug a running process by clicking Attach to a Process on the File menu or by pressing F6.

When the Attach to Process dialog box appears, select the Noninvasive check box. Then, select the line that contains the process ID and name that you want. (You can also enter the process ID in the Process ID box.) Finally, click OK.

Debugger Command Window

If the debugger is already active, you can noninvasively debug a running process by using the **.attach -v (Attach to Process)** command in the Debugger Command window.

You can use the .attach command if the debugger is already debugging one or more processes invasively. You can use this command in CDB if it is dormant, but not in a dormant WinDbg.

If the .attach -v command is successful, the debugger debugs the specified process the next time that the debugger issues an execution command. Because execution is not permitted during noninvasive debugging, the debugger cannot noninvasively debug more than one process at a time. This restriction also means that using the .attach -v command might make an existing invasive debugging session less useful.

Beginning the Debugging Session

For more information about how to begin a debugging session, see Debugger Operation.

Debugging in Assembly Mode

12/5/2018 • 4 minutes to read • Edit Online

If you have C or C++ source files for your application, you can use the debugger much more powerfully if you debug in source mode.

However, there are many times you cannot perform source debugging. You might not have the source files for your application. You might be debugging someone else's code. You might not have built your executable files with full .pdb symbols. And even if you can do source debugging on your application, you might have to trace Microsoft Windows routines that your application calls or that are used to load your application.

In these situations, you have to debug in assembly mode. Moreover, assembly mode has many useful features that are not present in source debugging. The debugger automatically displays the contents of memory locations and registers as they are accessed and displays the address of the program counter. This display makes assembly debugging a valuable tool that you can use together with source debugging.

Disassembly Code

The debugger primarily analyzes binary executable code. Instead of displaying this code in raw format, the debugger *disassembles* this code. That is, the debugger converts the code from machine language to assembly language.

You can display the resulting code (known as *disassembly code*) in several different ways:

- The **u (Unassemble)** command disassembles and displays a specified section of machine language.
- The **uf (Unassemble Function)** command disassembles and displays a function.
- The **up (Unassemble from Physical Memory)** command disassembles and displays a specified section of machine language that has been stored in physical memory.
- The **ur (Unassemble Real Mode BIOS)** command disassembles and displays a specified 16-bit real-mode code.
- The **ux (Unassemble x86 BIOS)** command disassembles and displays the x86-based BIOS code instruction set at a specified address.
- (WinDbg only) The disassembly window disassembles and displays a specified section of machine language. this window is automatically active if you select the **automatically open disassembly** command on the **window** menu. you can also open this window by clicking **disassembly** on the **view** menu, pressing alt+7, or pressing the **disassembly (alt+7)** button ( ) on the WinDbg toolbar.

The disassembly display appears in four columns: address offset, binary code, assembly language mnemonic, and assembly language details. The following example shows this display.

```
0040116b    45          inc         ebp
0040116c    fc          cld
0040116d    8945b0      mov         eax,[ebp-0x1c]
```

To the right of the line that represents the current program counter, the display shows the values of any memory locations or registers that are being accessed. If this line contains a branch instruction, the notation **[br=1]** or **[br=0]** appears. This notation indicates a branch that is or is not taken, respectively.

You can use the **.asm (Change Disassembly Options)** command to change how the disassembled instructions

are displayed.

In WinDbg's Disassembly window, the line that represents the current program counter is highlighted. Lines where breakpoints are set are also highlighted.

You can also use the following commands to manipulate assembly code:

- The **# (Search for Disassembly Pattern)** command searches a region of memory for a specific pattern. This command is equivalent to searching the four columns of the disassembly display.
- The **a (Assemble)** command can take assembly instructions and translate them into binary machine code.

Assembly Mode and Source Mode

The debugger has two different operating modes: *assembly mode* and *source mode*.

When you are single-stepping through an application, the size of a single step is one line of assembly code or one line of source code, depending on the mode.

Several commands create different data displays depending on the mode.

In WinDbg, the Disassembly window automatically moves to the foreground when you run or step through an application in assembly mode. In source mode, the Source window moves to the foreground.

To set the mode, you can do one of the following:

- Use the **l+, l- (Set Source Options)** command to control the mode. The **l-t** command activates assembly mode.
- (WinDbg only) Clear the **Source Mode** command on the **Debug** menu to cause the debugger to enter assembly mode.You can also click the **Source mode off** button ( ) on the toolbar.

In WinDbg, when you are in assembly mode, **ASM** appears visible on the status bar.

The shortcut menu in WinDbg's Disassembly window includes the **Highlight instructions from the current source line** command. This command highlights all of the instructions that correspond to the current source line. Frequently, a single source line corresponds to multiple assembly instructions. If code has been optimized, these assembly instructions might not be consecutive. The **Highlight instructions from the current source line** command enables you to find all of the instructions that were assembled from the current source line.

Assembly Language Source Files

If your application was written in assembly language, the disassembly that the debugger produces might not exactly match your original code. In particular, NO-OPs and comments will not be present.

If you want to debug your code by referencing the original .asm files, you must use source mode debugging. You can load the assembly file like a C or C++ source file. For more information about this kind of debugging, see Debugging in Source Mode.

Debugging in Source Mode

12/5/2018 • 7 minutes to read • Edit Online

Debugging an application is easier if you can analyze the source of the code, instead of the disassembled binaries.

WinDbg, CDB, and KD can use source code in debugging, if the source language is C, C++, or assembly.

Compilation Requirements

To use source debugging, you must have your compiler or linker create symbol files (.pdb files) when the binaries are built. These symbol files show the debugger how the binary instructions correspond to the source lines.

Also, the debugger must be able to access the actual source files , because symbol files do not contain the actual source text.

If it is possible, the compiler and linker should not optimize your code. Source debugging and access to local variables are more difficult, and sometimes almost impossible, if the code has been optimized. If you are using the Build utility as your compiler and linker, set the MSC_OPTIMIZATION macro to **/Od /Oi** to avoid optimization.

Locating the Symbol Files and Source Files

To debug in source mode, the debugger must be able to find the source files and the symbol files. For more information, see Source Code.

Beginning Source Debugging

The debugger can display source information whenever it has proper symbols and source files for the thread that is currently being debugged.

If you start a new user-mode application by using the debugger, the initial break occurs when Ntdll.dll loads the application. Because the debugger does not have access to the Ntdll.dll source files, you cannot access source information for your application at this point.

To move the program counter to the beginning of the application, add a breakpoint at the entry point to your binary. In the Debugger Command window, type the following command.

```
bp main
g
```

The application is then loaded and stops when the **main** function is entered. (Of course, you can use any entry point, not only **main**.)

If the application throws an exception, it breaks into the debugger. Source information is available at this point. However, if you issue a break by using the **CTRL+C**, CTRL+BREAK, or Debug | Break command, the debugger creates a new thread, so you cannot see your source code.

After you have reached a thread that you have source files for, you can use the Debugger Command window to execute source debugging commands. If you are using WinDbg, the Source window appears. If you have already opened a Source window by clicking **Open Source File** on the **File** menu, WinDbg typically create a new window for the source. You can close the previous window without affecting the debugging process.

Source Debugging in the WinDbg GUI

If you are using WinDbg, a Source window appears as soon as the program counter is in code that the debugger has source information for.

WinDbg displays one Source window for each source file that you or WinDbg opened. For more information about the text properties of this window, see Source Windows.

You can then step through your application or execute to a breakpoint or to the cursor. For more information about stepping and tracing commands, see Controlling the Target.

If you are in source mode, the appropriate Source window moves to the foreground as you step through your application. Because there are also Microsoft Windows routines that are called during the application's execution, the debugger might move a Disassembly window to the foreground when this kind of call occurs (because the debugger does not have access to the source for these functions). When the program counter returns to known source files, the appropriate Source window becomes active.

As you move through the application, WinDbg highlights your location in the Source window and the Disassembly window. Lines at which breakpoints are set are also highlighted. The source code is colored according to the parsing of the language. If the Source window has been selected, you can hover over a symbol with the mouse to evaluate it. For more information about these features and how to control them, see Source Windows.

To activate source mode in WinDbg, use the **l+t** command, click **source mode** on the **debug** menu, or click the **source mode on** button ( ) on the toolbar.

When source mode is active, the **ASM** indicator appears unavailable on the status bar.

You can view or alter the values of any local variables as you step through a function in source mode. For more information, see Reading and Writing Memory.

Source Debugging in the Debugger Command Window

If you are using CDB, you do not have a separate Source window. However, you can still view your progress as you step through the source.

Before you can do source debugging in CDB, you have to load source line symbols by issuing the **.lines (Toggle Source Line Support)** command or by starting the debugger with the **-lines command-line option**.

If you execute an **l+t** command, all program stepping is performed one source line at a time. Use **l-t** to step one assembly instruction at a time. If you are using WinDbg, this command has the same effect as selecting or clearing **Source Mode** on the **Debug** menu or using the toolbar buttons.

The **l+s** command displays the current source line and line number at the prompt. If you want to see only the line number, use **l+l** instead.

If you use **l+o** and **l+s**, only the source line is displayed while you step through the program. The program counter, disassembly code, and register information are hidden. This kind of display enables you to quickly step through the code and view nothing but the source.

You can use the **lsp (Set Number of Source Lines)** command to specify exactly how many source lines are displayed when you step through or execute the application.

The following sequence of commands is an effective way to step through a source file.

```
.lines        enable source line information
bp main       set initial breakpoint
l+t           stepping will be done by source line
l+s           source lines will be displayed at prompt
g             run program until "main" is entered
pr            execute one source line, and toggle register display off
p             execute one source line
```

Because **ENTER** repeats the last command, you can now step through the application by using the ENTER key. Each step causes the source line, memory offset, and assembly code to appear.

For more information about how to interpret the disassembly display, see Debugging in Assembly Mode.

When the assembly code is displayed, any memory location that is being accessed is displayed at the right end of the line. You can use the **d* (Display Memory)** and **e* (Enter Values)** commands to view or change the values in these locations.

If you have to view each assembly instruction to determine offsets or memory information, use **l-t** to step by assembly instructions instead of source lines. The source line information can still be displayed. Each source line corresponds to one or more assembly instructions.

All of these commands are available in WinDbg and in CDB. You can use the commands to view source line information from WinDbg's Debugger Command window instead of from the Source window.

Source Lines and Offsets

You can also perform source debugging by using the expression evaluator to determine the offset that corresponds to a specific source line.

The following command displays a memory offset.

```
? `[[module!]filename][:linenumber]`
```

If you omit *filename*, the debugger searches for the source file that corresponds to the current program counter.

The debugger reads *linenumber* as a decimal number unless you add **0x** before it, regardless of the current default radix. If you omit *linenumber*, the expression evaluates to the initial address of the executable file that corresponds to the source file.

This syntax is understood in CDB only if the **.lines** command or the **-lines** command-line option has loaded source line symbols.

This technique is very versatile, because you can use it regardless of where the program counter is pointing. For example, this technique enables you to set breakpoints in advance, by using commands such as the following.

```
bp `source.c:31`
```

For more information, see Source Line Syntax and Using Breakpoints.

Stepping and Tracing in Source Mode

When you are debugging in source mode, there can be multiple function calls on a single source line. You cannot use the **p** and **t** commands to separate these function calls.

For example, in the following command, the **t** command steps into both **GetTickCount** and **printf**, while the **p** command steps over both function calls.

```
printf( "%x\n", GetTickCount() );
```

If you want to step over certain calls while tracing into other calls, use **.step_filter (Set Step Filter)** to indicate which calls to step over.

You can use **_step_filter** to filter out framework functions (for example, Microsoft Foundation Classes (MFC) or Active Template Library (ATL) calls).

Debugging Optimized Code and Inline Functions

12/5/2018 • 7 minutes to read • Edit Online

For Windows 8, the debugger and the Windows compiler have been enhanced so that you can debug optimized code and debug inline functions. The debugger displays parameters and local variables regardless of whether they are stored in registers or on the stack. The debugger also displays inline functions in the call stack. For inline functions, the debugger displays local variables, but not parameters.

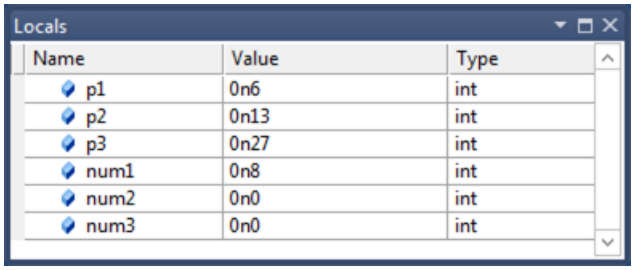
When code gets optimized, it is transformed to run faster and use less memory. Sometimes functions are removed as a result of dead code removal, code being merged, or functions being placed inline. Local variables and parameters can also be removed. Many code optimizations remove local variables that are not needed or used; other optimizations remove induction variables in loops. Common sub-expression elimination merges local variables together.

Retail builds of Windows are optimized. So if you are running a retail build of Windows, it is especially helpful to have a debugger that is designed to work well with optimized code. To make debugging of optimized code effective, two primary features are required: 1) accurate display of local variables, and 2) display of inline functions on the call stack.

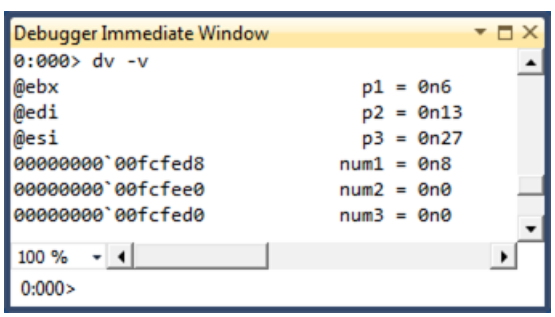
Accurate display of local variables and parameters

To facilitate the accurate display of local variables and parameters, the compiler records information about the locations of local variables and parameters in symbol (PDB) files. These location records track the variables' storage locations and the specific code ranges where these locations are valid. These records not only help track the locations (in registers or in stack slots) of the variables, but also the movement of the variables. For example, a parameter might first be in register RCX, but is moved to a stack slot to free up RCX, then moved to register R8 when it is heavily used in a loop, and then moved to different stack slot when the code is out of the loop. The Windows debugger consumes the rich location records in the PDB files and uses the current instruction pointer to select the appropriate location records for the local variables and parameters.

This screen shot of the Locals window in Visual Studio shows the parameters and local variables for a function in an optimized 64-bit application. The function is not inline, so we see both parameters and local variables.



You can use the **dv -v** command to see the locations of the parameters and local variables.



Notice that the Locals window displays the parameters correctly even though they are stored in registers.

In addition to tracking variables with primitive types, the location records track data members of local structures and classes. The following debugger output displays local structures.

```
0:000> dt My1
Local var Type _LocalStruct
   +0x000 i1               : 0n0 (edi)
   +0x004 i2               : 0n1 (rsp+0x94)
   +0x008 i3               : 0n2 (rsp+0x90)
   +0x00c i4               : 0n3 (rsp+0x208)
   +0x010 i5               : 0n4 (r10d)
   +0x014 i6               : 0n7 (rsp+0x200)

0:000> dt My2
Local var @ 0xefa60 Type _IntSum
   +0x000 sum1             : 0n4760 (edx)
   +0x004 sum2             : 0n30772 (ecx)
   +0x008 sum3             : 0n2 (r12d)
   +0x00c sum4             : 0n0
```

Here are some observations about the preceding debugger output.

- The local structure **My1** illustrates that the compiler can spread local structure data members to registers and non-contiguous stack slots.
- The output of the command **dt My2** will be different from the output of the command **dt _IntSum 0xefa60**. You cannot assume that the local structure will occupy a contiguous block of stack memory. In the case of **My2**, only `sum4` stays in the original stack block; the other three data members are moved to registers.
- Some data members can have multiple locations. For example, **My2.sum2** has two locations: one is register ECX (which the Windows debugger chooses) and the other is 0xefa60+0x4 (the original stack slot). This could happen for primitive-type local variables also, and the Windows debugger imposes precedent heuristics to determine which location to use. For example, register locations always trump stack locations.

Display of inline functions on the call stack

During code optimization, some functions are placed in line. That is, the body of the function is placed directly in the code like a macro expansion. There is no function call and no return to the caller. To facilitate the display of inline functions, the compiler stores data in the PDB files that helps decode the code chunks for the inline functions (that is, sequences of code blocks in caller functions that belong to the callee functions that are being placed inline) as well as the local variables (scoped local variables in those code blocks). This data helps the debugger include inline functions as part of the stack unwind.

Suppose you compile an application and force a function named `func1` to be inline.

```
__forceinline int func1(int p1, int p2, int p3)
{
   int num1 = 0;
   int num2 = 0;
   int num3 = 0;
   ...
}
```

You can use the **bm** command to set a breakpoint at `func1`.

```
0:000> bm MyApp!func1
  1: 000007f6`8d621088 @!"MyApp!func1" (MyApp!func1 inlined in MyApp!main+0x88)
0:000> g

Breakpoint 1 hit
MyApp!main+0x88:
000007f6`8d621088 488d0d21110000  lea     rcx,[MyApp!`string' (000007f6`8d6221b0)]
```

After you take one step into `func1`, you can use the **k** command to see `func1` on the call stack. You can use the **dv** command to see the local variables for `func1`. Notice that the local variable `num3` is shown as unavailable. A local variable can be unavailable in optimized code for a number of reasons. It might be that the variable doesn't exist in the optimized code. It might be that the variable has not been initialized yet or that the variable is no longer being used.

```
0:000> p
MyApp!func1+0x7:
000007f6`8d62108f 8d3c33          lea     edi,[rbx+rsi]

0:000> knL
# Child-SP          RetAddr           Call Site
00 (Inline Function) --------`-------- MyApp!func1+0x7
01 00000000`0050fc90 000007f6`8d6213f3 MyApp!main+0x8f
02 00000000`0050fcf0 000007ff`c6af0f7d MyApp!__tmainCRTStartup+0x10f
03 00000000`0050fd20 000007ff`c7063d6d KERNEL32!BaseThreadInitThunk+0xd
04 00000000`0050fd50 00000000`00000000 ntdll!RtlUserThreadStart+0x1d

0:000> dv -v
00000000`0050fcb0            num1 = 0n0
00000000`0050fcb4            num2 = 0n0
<unavailable>                num3 = <value unavailable>
```

If you look at frame 1 in the stack trace, you can see the local variables for the `main` function. Notice that two of the variables are stored in registers.

```
0:000> .frame 1
01 00000000`0050fc90 000007f6`8d6213f3 MyApp!main+0x8f

0:000> dv -v
00000000`0050fd08               c = 0n7
@ebx                            b = 0n13
@esi                            a = 0n6
```

The Windows debugger aggregates data from PDB files to find all the places where a specific function has been placed inline. You can use the **x** command to list all the caller sites of the an inline function.

```
0:000> x simple!MoreCalculate
00000000`ff6e1455 simple!MoreCalculate =  (inline caller) simple!wmain+8d
00000000`ff6e1528 simple!MoreCalculate =  (inline caller) simple!wmain+160

0:000> x simple!Calculate
00000000`ff6e141b simple!Calculate =  (inline caller) simple!wmain+53
```

Because the Windows debugger can enumerate all the caller sites of an inline function, it can set a breakpoints inside the inline function by calculating the offsets from the caller sites. You can use the **bm** command (which is used to set breakpoints that match regular expression patterns) to set breakpoints for inline functions.

The Windows debugger groups all breakpoints that are set for a specific inline function into a breakpoint container. You can manipulate the breakpoint container as a whole by using commands like **be**, **bd**, **bc**. See the

following **bd 3** and **bc 3** command examples. You can also manipulate individual breakpoints. See the following **be 2** command example.

```
0:000> bm simple!MoreCalculate
  2: 00000000`ff6e1455 @!"simple!MoreCalculate" (simple!MoreCalculate inlined in simple!wmain+0x8d)
  4: 00000000`ff6e1528 @!"simple!MoreCalculate" (simple!MoreCalculate inlined in simple!wmain+0x160)

0:000> bl
 0 e 00000000`ff6e13c8 [n:\win7\simple\simple.cpp @ 52]    0001 (0001)  0:**** simple!wmain
 3 e <inline function>     0001 (0001)  0:**** {simple!MoreCalculate}
     2 e 00000000`ff6e1455 [n:\win7\simple\simple.cpp @ 58]    0001 (0001)  0:**** simple!wmain+0x8d (inline
function simple!MoreCalculate)
     4 e 00000000`ff6e1528 [n:\win7\simple\simple.cpp @ 72]    0001 (0001)  0:**** simple!wmain+0x160 (inline
function simple!MoreCalculate)

0:000> bd 3
0:000> be 2

0:000> bl
 0 e 00000000`ff6e13c8 [n:\win7\simple\simple.cpp @ 52]    0001 (0001)  0:**** simple!wmain
 3 d <inline function>     0001 (0001)  0:**** {simple!MoreCalculate}
     2 e 00000000`ff6e1455 [n:\win7\simple\simple.cpp @ 58]    0001 (0001)  0:**** simple!wmain+0x8d (inline
function simple!MoreCalculate)
     4 d 00000000`ff6e1528 [n:\win7\simple\simple.cpp @ 72]    0001 (0001)  0:**** simple!wmain+0x160 (inline
function simple!MoreCalculate)

0:000> bc 3

0:000> bl
 0 e 00000000`ff6e13c8 [n:\win7\simple\simple.cpp @ 52]    0001 (0001)  0:**** simple!wmain
```

Because there are no explicit call or return instructions for inline functions, source-level stepping is especially challenging for a debugger. For example, you could unintentionally step in to an inline function (if the next instruction is part of an inline function), or you could step in and step out of the same inline function multiple times (because the code blocks for the inline function have been split and moved by the compiler). To preserve the familiar stepping experience, the Windows debugger maintains a small conceptual call stack for every code instruction address and builds an internal state machine to execute step-in, step-over, and step-out operations. This gives a reasonably accurate approximation to the stepping experience for non-inline functions.

Additional Information

Note You can use the **.inline 0** command to disable inline function debugging. The **.inline 1** command enables inline function debugging. Standard Debugging Techniques

Related topics

Standard Debugging Techniques

Debugging Managed Code Using the Windows Debugger

12/5/2018 • 6 minutes to read • Edit Online

You can use the Windows debuggers (WinDbg, CDB, and NTSD) to debug target applications that contain managed code. To debug managed code, you must load the SOS debugging extension (sos.dll) and a data access component (mscordacwks.dll).

The Windows debuggers are separate from the Visual Studio debugger. For information about the distinction between the Windows debuggers and the Visual Studio debugger, see Windows Debugging.

Introduction to Managed Code

Managed code is executed together with the Microsoft .NET Common Language Runtime (CLR). In a managed-code application, the binary code that the compiler produces is in Microsoft Intermediate Language (MSIL), which is platform-independent.

When managed code is run, the runtime produces native code that is platform-specific. The process of generating native code from MSIL is called *just-in-time (JIT) compiling*. After the JIT compiler has compiled the MSIL for a specific method, the method's native code remains in memory. Whenever this method is later called, the native code executes and the JIT compiler does not have to be involved.

You can build managed code by using several compilers that are manufactured by a variety of software producers. In particular, Microsoft Visual Studio can build managed code from several different languages including C#, Visual Basic, JScript, and C++ with managed extensions.

The CLR is not updated every time the .NET Framework is updated. For example, versions 2.0, 3.0, and 3.5 of the .NET Framework all use version 2.0 of the CLR. The following table shows the version and filename of the CLR used by each version of the .NET Framework.

.NET FRAMEWORK VERSION	CLR VERSION	CLR FILENAME
1.1	1.1	mscorwks.dll
2.0	2.0	mscorwks.dll
3.0	2.0	mscorwks.dll
3.5	2.0	mscorwks.dll
4.0	4.0	clr.dll
4.5	4.0	clr.dll

Debugging Managed Code

To debug managed code, the debugger must load these two components.

- Data access component (DAC) (mscordacwks.dll)
- SOS debugging extension (sos.dll)

Note For all versions of the .NET Framework, the filename of the DAC is mscordacwks.dll, and the filename of the SOS debugging extension is sos.dll.

Getting the SOS Debugging Extension (sos.dll)

The SOS debugging extension (sos.dll) files are not included in the current version of Debugging Tools for Windows.

For .NET Framework versions 2.0 and later, sos.dll is included in the .NET Framework installation.

For version 1.*x* of the .NET Framework, sos.dll is not included in the .NET Framework installation. To get sos.dll for .NET Framework 1.*x*, download the 32-bit version of Windows 7 Debugging Tools for Windows.

Windows 7 Debugging Tools for Windows is included in the Windows SDK for Windows 7, which is available at these two places:

- Windows SDK for Windows 7 and .NET Framework 4.0
- Windows SDK for Windows 7 and .NET Framework 4.0 (ISO)

If you are running an x64 version of Windows, use the ISO site, so that you can specify that you want the 32-bit version of the SDK. Sos.dll is included only in the 32-bit version of Windows 7 Debugging Tools for Windows.

Loading mscordacwks.dll and sos.dll (live debugging)

Assume that the debugger and the application being debugged are running on the same computer. Then the .NET Framework being used by the application is installed on the computer and is available to the debugger.

The debugger must load a version of the DAC that is the same as the version of the CLR that the managed-code application is using. The bitness (32-bit or 64-bit) must also match. The DAC (mscordacwks.dll) comes with the .NET Framework. To load the correct version of the DAC, attach the debugger to the managed-code application, and enter this command.

.cordll -ve -u -l

The output should be similar to this.

```
CLRDLL: Loaded DLL C:\Windows\Microsoft.NET\Framework64\v4.0.30319\mscordacwks.dll
CLR DLL status: Loaded DLL C:\Windows\Microsoft.NET\Framework64\v4.0.30319\mscordacwks.dll
```

To verify that the version of mscordacwks.dll matches the version of the CLR that the application is using, enter one of the following commands to display information about the loaded CLR module.

lmv mclr (for version 4.0 of the CLR)

lmv mscorwks (for version 1.0 or 2.0 of the CLR)

The output should be similar to this.

```
start             end                 module name
000007ff`26710000 000007ff`2706e000   clr        (deferred)
    Image path: C:\Windows\Microsoft.NET\Framework64\v4.0.30319\clr.dll
...
```

In the preceding example, notice that the version of the CLR (clr.dll) matches the version of the DAC (mscordacwks.dll): v4.0.30319. Also notice that both components are 64-bit.

When you use **.cordll** to load the DAC, the SOS debugging extension (sos.dll) might get loaded automatically. If sos.dll doesn't get loaded automatically, you can use one of these commands to load it.

.loadby sos clr (for version 4.0 of the CLR)

.loadby sos mscorwks (for version 1.0 or 2.0 of the CLR)

As an alternative to using **.loadby**, you can use **.load**. For example, to load version 4.0 of the 64-bit CLR, you could enter a command similar to this.

.load C:\Windows\Microsoft.NET\Framework64\v4.0.30319\sos.dll

In the preceding output, notice that the version of the SOS debugging extension (sos.dll) matches the version of the CLR and the DAC: v4.0.30319. Also notice that all three components are 64-bit.

Loading mscordacwks.dll and sos.dll (dump file)

Suppose you use the debugger to open a dump file (of a managed-code application) that was created on another computer.

The debugger must load a version of the DAC that is the same as the version of the CLR that the managed-code application was using on the other computer. The bitness (32-bit or 64-bit) must also match.

The DAC (mscordacwks.dll) comes with the .NET Framework, but let's assume that you do not have the correct version of the .NET Framework installed on the computer that is running the debugger. You have three options.

- Load the DAC from a symbol server. For example, you could include Microsoft's public symbol server in your symbol path.
- Install the correct version of the .NET Framework on the computer that is running the debugger.
- Get the correct version of mscordacwks.dll from the person who created the dump file (on another computer) and manually copy it to the computer that is running the debugger.

Here we illustrate using Microsoft's public symbol server.

Enter these commands.

.sympath+ srv* (Add symbol server to symbol path.)

!sym noisy

.cordll -ve -u -l

The output will be similar to this.

```
CLRDLL: Unable to get version info for 'C:\Windows\Microsoft.NET
   \Framework64\v4.0.30319\mscordacwks.dll', Win32 error 0n87

SYMSRV:  C:\ProgramData\dbg\sym\mscordacwks_AMD64_AMD64_4.0.30319.18010.dll
   \5038768C95e000\mscordacwks_AMD64_AMD64_4.0.30319.18010.dll not found

SYMSRV:  mscordacwks_AMD64_AMD64_4.0.30319.18010.dll from
   https://msdl.microsoft.com/download/symbols: 570542 bytes - copied
...
SYMSRV:  C:\ProgramData\dbg\sym\SOS_AMD64_AMD64_4.0.30319.18010.dll
   \5038768C95e000\SOS_AMD64_AMD64_4.0.30319.18010.dll not found

SYMSRV:  SOS_AMD64_AMD64_4.0.30319.18010.dll from
   https://msdl.microsoft.com/download/symbols: 297048 bytes - copied
...
Automatically loaded SOS Extension
...
```

In the preceding output, you can see that the debugger first looked for mscordacwks.dll and sos.dll on the local computer in C:\Windows\Microsoft.NET and in the symbol cache (C:\ProgramData\dbg\sym). When the debugger did not find the correct versions of the files on the local computer, it retrieved them from the public symbol server.

To verify that the version of mscordacwks.dll matches the version of the CLR that the application was using, enter one of the following commands to display information about the loaded CLR module.

lmv -mclr (for version 4.0 of the CLR)

lmv -mscorwks (for version 1.0 or 2.0 of the CLR)

The output should be similar to this.

```
start             end                 module name
000007ff`26710000 000007ff`2706e000   clr        (deferred)
    Image path: C:\Windows\Microsoft.NET\Framework64\v4.0.30319\clr.dll
...
```

In the preceding example, notice that the version of the CLR (clr.dll) matches the version of the DAC (mscordacwks.dll): v4.0.30319. Also notice that both components are 64-bit.

Using the SOS Debugging Extension

To verify that the SOS debugging extension loaded correctly, enter the **.chain** command.

```
0:000> .chain
Extension DLL search Path:
...
Extension DLL chain:
    C:\ProgramData\dbg\sym\SOS_AMD64_AMD64_4.0.30319.18010.dll\...
        ...
    dbghelp: image 6.13.0014.1665, API 6.2.6, built Wed Dec 12 03:02:43 2012
...
```

To test the SOS debugging extension, enter **!sos.help**. Then try one of the command provided by the SOS debugging extension. For example, you could try **!sos.DumpDomain** or the **!sos.Threads** command.

Notes

Sometimes a managed-code application loads more than one version of the CLR. In that case, you must specify which version of the DAC to load. For more information, see **.cordll**.

Debugging Windows Apps Using the Windows Debugger

12/5/2018 • 2 minutes to read • Edit Online

You can use the Windows debuggers (WinDbg, CDB, and NTSD) to debug Windows apps. Use the **PLMDebug** tool to take control of suspending, resuming, and terminating a Windows app while you are debugging.

To debug a managed-code Windows app, load the SOS debugging extension (sos.dll) and a data access component (mscordacwks.dll). For more information, see Debugging Managed Code Using the Windows Debugger.

The windows debuggers are separate from the Visual Studio debugger. For information about the distinction between the windows debuggers and the Visual Studio debugger, see Windows Debugging.

Changing Contexts

6/25/2019 • 6 minutes to read • Edit Online

In kernel-mode debugging, there are many processes, threads, and sometimes user sessions that are executing at the same time. Therfore, phrases such as "virtual address 0x80002000" or "the **eax** register" are ambiguous. You must specify the *context* in which such phrases can be understood.

The debugger has five different contexts that you can set while you are debugging:

1. The session context indicates the default user session.
2. The process context determines how the debugger interprets virtual addresses.
3. The *user-mode address context* is almost never set directly. This context is automatically set when you change the process context.
4. The register context determines how the debugger interprets registers and also controls the results of a stack trace. This context is also known as the *thread context*, although that term is not completely accurate. An *explicit context* is also a type of register context. If you specify an explicit context, that context is used instead of the current register context.
5. The local context determines how the debugger interprets local variables. This context is also known as the *scope*.

Session Context

Multiple logon sessions can run at the same time. Each logon session has its own processes.

The **!session** extension displays all logon sessions or changes the current session context.

The session context is used by the **!sprocess** and **!spoolused** extensions when the session number is entered as "-2".

When the session context is changed, the process context is automatically changed to the active process for that session.

Process Context

Each process has its own page directory that records how virtual addresses are mapped to physical addresses. When any thread within a process is executing, the Windows operating system uses this page directory to interpret virtual addresses.

During user-mode debugging, the current process determines the process context. Virtual addresses that are used in debugger commands, extensions, and debugging information windows are interpreted by using the page directory of the current process.

During kernel-mode debugging, you can set the process context by using the **.process (Set Process Context)** command. Use this command to select which process's page directory is used to interpret virtual addresses. After you set the process context, you can use this context in any command that takes addresses. You can even set breakpoints at this address. By including a **/i** option in the **.process** command to specify invasive debugging, you can also use the kernel debugger to set breakpoints in user space.

You can also set user-mode breakpoints from the kernel debugger by using a process-specific breakpoint on a kernel-space function. Set strategic breakpoints and wait for the appropriate context to come up.

The *user-mode address context* is part of the process context. Typically, you do not have to set the user-mode address context directly. If you set the process context, the user-mode address context automatically changes

to the directory base of the relevant page table for the process.

When you set the process context during kernel-mode debugging, that process context is retained until another **.process** command changes the context. The user-mode address context is also retained until a **.process** or **.context** command changes it. These contexts are not changed when the target computer executes, and they are not affected by changes to the register context or the local context.

Register Context

Each thread has its own register values. These values are stored in the CPU registers when the thread is executing and are stored in memory when another thread is executing.

During user-mode debugging, the current thread typically determines the register context. Any reference to registers in debugger commands, extensions, and debugging information windows is interpreted according to the current thread's registers.

You can change the register context to a value other than the current thread while you are performing user-mode debugging by using one of the following commands:

.cxr (Display Context Record)

.ecxr (Display Exception Context Record)

During kernel-mode debugging, you can control the register context by using a variety of debugger commands, including the following commands:

.thread (Set Register Context)

.cxr (Display Context Record)

.trap (Display Trap Frame)

These commands do not change the values of the CPU registers. Instead, the debugger retrieves the specified register context from its location in memory. Actually, the debugger can retrieve only the *saved* register values. (Other values are set dynamically and are not saved. The saved values are sufficient to re-create a stack trace.

After the register context is set, the new register context is used for any commands that use register values, such as **k (Display Stack Backtrace)** and **r (Registers)**.

However, when you are debugging multiprocessor computers, some commands enable you to specify a processor. (For more information about such commands, see Multiprocessor Syntax.) If you specify a processor for a command, the command uses the register context of the active thread on the specified processor instead of the current register context, even if the specified processor is the currently-active processor.

Also, if the register context does not match the current processor mode setting, these commands produce incorrect or meaningless output. To avoid the output errors, commands that depend on the register state fail until you change the processor mode to match the register context. To change the processor mode, use the **.effmach (Effective Machine)** command,

Changing the register context can also change the local context. In this manner, the register context can affect the display of local variables.

If any application execution, stepping, or tracing occurs, the register context is immediately reset to match the program counter's position. In user mode, the register context is also reset if the current process or thread is changed.

The register context affects stack traces, because the stack trace begins at the location that the stack pointer register (**esp** on an x86-based processor) points to. If the register context is set to an invalid or inaccessible

value, stack traces cannot be obtained.

You can apply a processor breakpoint (data breakpoint) to a specific register context by using the **.apply_dbp (Apply Data Breakpoint to Context)** command.

Local Context

When a program is executing, the meaning of local variables depends on the location of the program counter, because the scope of such variables extends only to the function that they are defined in.

When you are performing user-mode or kernel-mode debugging, the debugger uses the scope of the current function (the current frame on the stack) as the local context. To change this context, use the **.frame (Set Local Context)** command, or double-click the desired frame in the Calls window.

In user-mode debugging, the local context is always a frame within the stack trace of the current thread. In kernel-mode debugging, the local context is always a frame within the stack trace of the current register context's thread.

You can use only one stack frame at a time for the local context. Local variables in other frames cannot be accessed.

The local context is reset if any of the following events occur:

- Any program execution, stepping or tracing
- Any use of the thread delimiter (~) in any command
- Any change to the register context

The **!for_each_frame** extension enables you to execute a single command repeatedly, once for each frame in the stack. This command changes the local context for each frame, executes the specified command, and then returns the local context to its original value.

Controlling Processes and Threads

12/5/2018 • 3 minutes to read • Edit Online

When you are performing user-mode debugging, you activate, display, freeze, unfreeze, suspend, and unsuspend processes and threads.

The *current* or *active* process is the process that is currently being debugged. Similarly, the *current* or *active* thread is the thread that the debugger is currently controlling. The actions of many debugger commands are determined by the identity of the current process and thread. The current process also determines the virtual address mappings that the debugger uses.

When debugging begins, the current process is the one that the debugger is attached to or that caused the exception that broke into the debugger. Similarly, the current thread is the one that was active when the debugger attached to the process or that caused the exception. However, you can use the debugger to change the current process and thread and to freeze or unfreeze individual threads.

In kernel-mode debugging, processes and threads are not controlled by the methods that are described in this section. For more information about how processes and threads are manipulated in kernel mode, see Changing Contexts.

Displaying Processes and Threads

To display process and thread information, you can use the following methods:

- The **| (Process Status)** command
- The **~ (Thread Status)** command
- (WinDbg only) The Processes and Threads window

Setting the Current Process and Thread

To change the current process or thread, you can use the following methods:

- The **|s (Set Current Process)** command
- The **~s (Set Current Thread)** command
- (WinDbg only) The Processes and Threads window

Freezing and Suspending Threads

The debugger can change the execution of a thread by *suspending* the thread or by *freezing* the thread. These two actions have somewhat different effects.

Each thread has a *suspend count* that is associated with it. If this count is one or larger, the system does not run the thread. If the count is zero or lower, the system runs the thread when appropriate.

Typically, each thread has a suspend count of zero. When the debugger attaches to a process, it increments the suspend counts of all threads in that process by one. If the debugger detaches from the process, it decrements all suspend counts by one. When the debugger executes the process, it temporarily decrements all suspend counts by one.

You can control the suspend count of any thread from the debugger by using the following methods:

- The **~n (Suspend Thread)** command increments the specified thread's suspend count by one.
- The **~m (Resume Thread)** command decrements the specified thread's suspend count by one.

The most common use for these commands is to raise a specific thread's suspend count from one to two. When the debugger executes or detaches from the process, the thread then has a suspend count of one and remains suspended, even if other threads in the process are executing.

You can suspend threads even when you are performing noninvasive debugging.

The debugger can also *freeze* a thread. This action is similar to suspending the thread in some ways. However, "frozen" is only a debugger setting. Nothing in the Windows operating system recognizes that anything is different about this thread.

By default, all threads are unfrozen. When the debugger causes a process to execute, threads that are frozen do not execute. However, if the debugger detaches from the process, all threads unfreeze.

To freeze and unfreeze individual threads, you can use the following methods:

- The **~f (Freeze Thread)** command freezes the specified thread.
- The **~u (Unfreeze Thread)** command unfreezes the specified thread.

In any event, threads that belong to the target process never execute when the debugger has broken into the target. The suspend count of a thread affects the thread's behavior only when the debugger executes the process or detaches. The frozen status affects the thread's behavior only when the debugger executes the process.

Threads and Processes in Other Commands

You can add thread specifiers or process specifiers before many other commands. For more information, see the individual command topics.

You can add the **~e (Thread-Specific Command)** qualifier before many commands and extension commands. This qualifier causes the command to be executed with respect to the specified thread. This qualifier is especially useful if you want to apply a command to more than one thread. For example, the following command repeats the **!gle** extension command for every thread that is being debugged.

```
~*e !gle
```

Multiple Systems

The debugger can attach to multiple targets at the same time. When these processes include dump files or include live targets on more than one computer, the debugger references a system, process, and thread for each action. For more information about this kind of debugging, see Debugging Multiple Targets.

Using Debugger Markup Language

12/5/2018 • 4 minutes to read • Edit Online

Debugger commands can provide output in plain text or in an enhanced format that uses Debugger Markup Language (DML). Output that is enhanced with DML includes links that you can click to execute related commands.

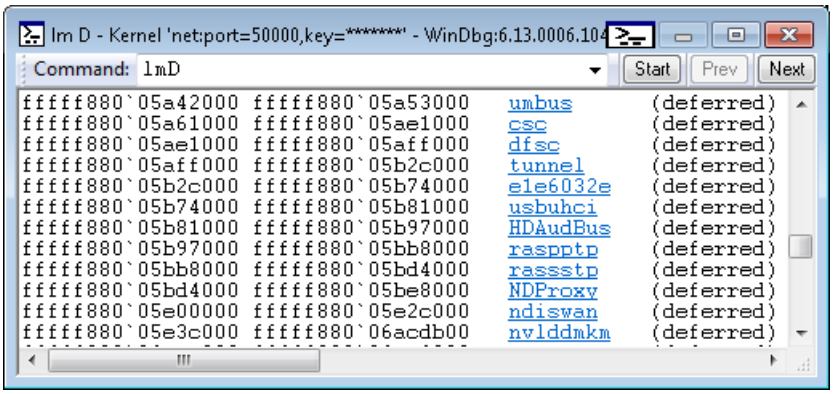
DML is available in Windows 10 and later.

DML Capable Commands

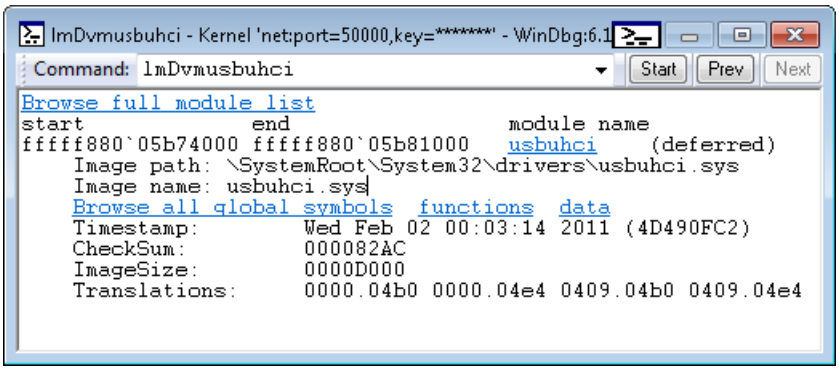
The following commands are capable of generating DML output:

- **.dml_start**
- **.dml_flow**
- **!dml_proc**
- **lmD**
- **kM**
- **.chain /D**
- **.help /D**
- **.printf /D**

The **lmD** command is an example of a command that is capable of providing DML output. The **lmD** command displays a list of loaded modules. As the following image shows, each module name is a link that you can click to get more detailed information about the module.



The following image shows the result of clicking the **usbuhci** link. The output includes additional links that enable you to explore further details of the usbuhci module.



Turning DML On and Off

The **.prefer_dml** command turns DML on or off. When DML is turned on (.prefer_dml 1), commands that are capable of generating DML output will generate DML output by default.

Console Enhancements

All of the Windows debuggers now have command output areas which support DML parsing. In windbg the command window supports all DML behavior and will show colors, font styles and links. The console debuggers, ntsd, cdb and kd, only support the color attributes of DML, and the only when running in a true console with color mode enabled. Debuggers with redirected I/O, ntsd –d or remote.exe sessions will not display any colors.

Console Debugger Color Mode

The console debuggers, ntsd, cdb and kd now have the ability to display colored output when running in a true console. This is not the default, it requires color mode to be explicitly enabled via tools.ini. The new col_mode <true|false> token in tools.ini controls the color mode setting. For more information about working with the tools.ini file, see Configuring tools.ini

When color mode is enabled the debugger can produce colored output. By default most colors are not set and instead default to the current console colors.

Windbg Command Browser Window

In Windows 10 and later Windbg the command browser window parses and displays DML. All tags such as <link>, <exec> and appearance modifications, are fully supported.

To start a command browser session using the menu in WinDbg, select **View**, **Command Browser**. The .browse <command> in the command window will open a new command browser window and execute the given command. For more information see Using the Command Browser Window in WinDbg. A new command browser window can also be opened with Ctrl+N.

The command browser window deliberately mimics the behavior of a web browser, with a drop-down history and previous/next buttons. The history drop-down only displays the last twenty commands but full history is kept so by going back in the commands you can get the drop-down to display older history.

You can have as many command windows open at once as you like. Command windows persist in workspaces but only save the current command; the history is not kept.

The WinDbg **View** menu has a **Set Browser Start Command** option which allows a user to set a preferred command for new browser windows to start with, such as .dml_start. This command is saved in workspaces.

A **Recent Commands** sub-window is available on the **View** menu to hold commands of interest. Selecting a recent command opens a new browser with the given command. There is a menu item on the browser window's context menu that adds the window's current command to the list of recent commands. The list of recent commands is persisted in workspaces.

The command browser window executes the command synchronously and so does not display output until the command has completed. Long-running commands will not show anything until they have finished.

Links have a right-click context menu similar to the right-click context menu in a web browser. Links can be opened in a new browser window. A link's command can be copied to the clipboard for use.

Clicking the icon near the upper-right corner of the title bar to set the command browser windows to either auto-refresh or manual-refresh. Auto-refresh browsers will automatically re-run their command on debugger state changes. This keeps the output live but at the cost of executing the command on all changes. Auto-refresh is on by default. If the browser does not need to be live the window's context menu can be used to disable auto-refresh.

Because commands are executed by the engine, not by the user interface, user-interface specific commands, such as **.cls (Clear Screen)**, will return a syntax error in when used in command browser windows. It also

means that when the user interface is a remote client, the command will be executed by the server, not by the client, and the command output will show server state.

Command browser windows can run any debugger command, it does not have to be a command that produces DML. You can use browser windows to have an arbitrary set of commands active for use.

Customizing DML

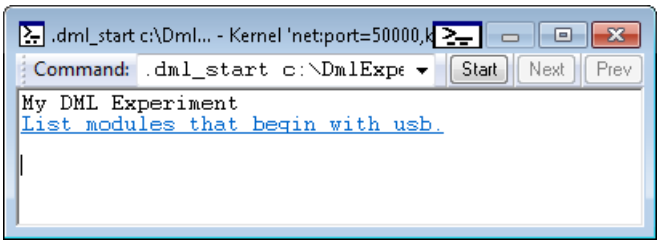
DML defines a small set of tags that can be included in command output. One example is the <link> tag. You can experiment with the <link> tag (and other DML tags) by using the **.dml_start** and **.browse** commands. The command **.browse .dml_start** *filepath* executes the commands stored in a DML file. The output is displayed in the Command Browser window instead of the regular command window.

Suppose the file c:\DmlExperiment.txt contains the following lines.

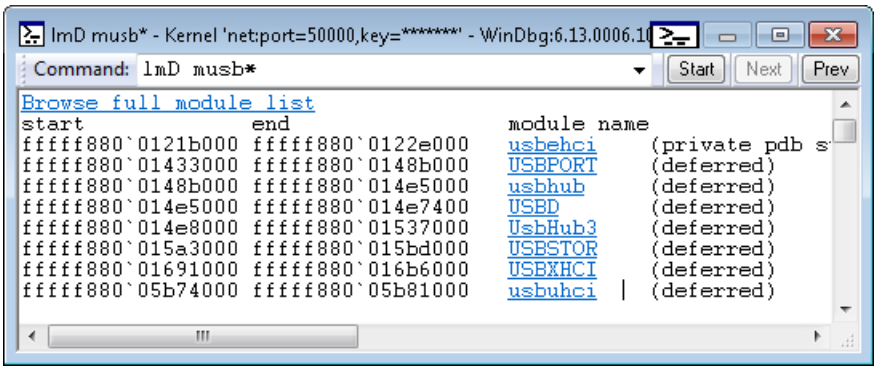
```
My DML Experiment
<link cmd="lmD musb*">List modules that begin with usb.</link>
```

The following command displays the text and link in the Command Browser window.

```
.browse .dml_start c:\Dml_Experiment.txt
```



If you click the **List modules that begin with usb** link, you see output similar to the following image.



For a thorough discussion of DML customization and a complete list of DML tags, see Customizing Debugger Output Using DML.

Controlling Exceptions and Events

3/6/2019 • 12 minutes to read • Edit Online

You can catch and handle exceptions in user-mode and kernel-mode applications by a variety of methods. An active debugger, a postmortem debugger, or an internal error handling routine are all common ways to handle exceptions.

For more information about the precedence order of these various exception handlers, see Enabling Postmortem Debugging.

When the Microsoft Windows operating system allows a debugger to handle an exception, the application that generated the exception *breaks into* the debugger. That is, the application stops and the debugger becomes active. The debugger can then handle the exception in some way or analyze the situation. The debugger can then end the process or let it resume running.

If the debugger ignores the exception and lets the application continue running, the operating system looks for other exception handlers as if no debugger was present. If the exception is handled, the application continues running. However, if the exception remains unhandled, the debugger is then given a second opportunity to deal with the situation.

Using the Debugger to Analyze an Exception

When an exception or event breaks into the debugger, you can use the debugger to examine the code that is being executed and the memory that the application is using. By altering certain quantities or jumping to a different point in the application, you might be able to remove the cause of the exception.

You can resume execution by issuing a **gh (Go with Exception Handled)** or **gn (Go with Exception Not Handled)** command.

If you issue the **gn** command in the debugger's second opportunity to handle the exception, the application ends.

Kernel-Mode Exceptions

Exceptions that occur in kernel-mode code are more serious than user-mode exceptions. If kernel-mode exceptions are not handled, a bug check is issued and the system stops.

As with user-mode exceptions, if a kernel-mode debugger is attached to the system, the debugger is notified before the *bug check screen* (also known as a *blue screen*) appears. If no debugger is attached, the bug check screen appears. In this case, the operating system might create a crash dump file.

Controlling Exceptions and Events from the Debugger

You can configure the debugger to react to specified exceptions and events in a specific way.

The debugger can set the break status for each exception or event:

- The event can cause a break into the debugger as soon as it occurs (the "first chance").
- The event can break in after other error handlers have been given an opportunity to respond (the "second chance").
- The event can also send the debugger a message but continue executing.
- The debugger can ignore the event.

The debugger can also set the handling status for each exception and event. The debugger can treat the event

like a handled exception or an unhandled exception. (Of course, events that are not actually errors do not require any handling.)

You can control the break status and handling status by doing one of the following:

- Use the **SXE**, **SXD**, **SXN**, or **SXI** command in the Debugger Command window.
- (CDB and NTSD) Use the **-x**, **-xe**, **-xd**, **-xn**, or **-xi** option on the **command line**.
- (CDB, NTSD, and KD) Use the **sxe** or **sxd** keyword in the Tools.ini file.
- (WinDbg only) Click Event Filters on the **Debug** menu to open the **Event Filters** dialog box, and then choose the options that you want.

The **SX*** command, the **-x*** command-line option, and the **sx*** Tools.ini keyword typically set the break status of the specified event. You can add the **-h** option to cause the handling status to be set instead.

There are four special event codes (**cc**, **hc**, **bpec**, and **ssec**) that always specify handling status instead of break status.

You can display the most recent exception or event by using the **.lastevent (Display Last Event)** command.

Controlling Break Status

When you set the break status of an exception or event, you can use the following options.

COMMAND	STATUS NAME	DESCRIPTION
SXE or **-xe**	**Break** (**Enabled**)	When this exception occurs, the target immediately breaks into the debugger. This break in occurs before any other error handlers are activated. This method is called *first-chance handling*.
SXD or **-xd**	**Second chance break** (**Disabled**)	The debugger does not break in for this kind of first-chance exception (although a message is displayed). If other error handlers cannot address this exception, execution stops and the target breaks into the debugger. This method is called *second-chance handling*.
SXN or **-xn**	**Output** (**Notify**)	When this exception occurs, the target application does not break into the debugger at all. However, a message is displayed that informs the user of this exception.
SXI or **-xi**	**Ignore**	When this exception occurs, the target application does not break into the debugger, and no message is displayed.

If an exception is not anticipated by an **SX*** setting, the target application breaks into the debugger on the second chance. The default status for events is listed in the following "Event Definitions and Defaults" section

of this topic.

To set break status by using the WinDbg graphical interface, Event Filters on the **Debug** menu, click the event that you want from the list in the **Event Filters** dialog box, and then select **Enabled**, **Disabled**, **Output**, or **Ignore**.

Controlling Handling Status

All events are considered unhandled, unless you use the **gh (Go with Exception Handled)** command.

All exceptions are considered unhandled, unless you use the **sx*** command together with the **-h** option.

Additionally, **SX*** options can configure the handling status for invalid handles, STATUS_BREAKPOINT break instructions, and single-step exceptions. (This configuration is separate from their break configuration.) When you configure their break status, these events are named **ch**, **bpe**, and **sse**, respectively. When you configure their handling status, these events are named **hc**, **bpec**, and **ssec**, respectively. (For the full listing of events, see the following "Event Definitions and Defaults" section.)

You can configure the handling status for the CTRL+C event (**cc**), but not its break status. If an application receives a CTRL+C event, the application always breaks into the debugger.

When you use the **SX*** command on **cc**, **hc**, **bpec**, and **ssec** events, or when you use the **SX*** command together with the **-h** option on an exception, the following actions occur.

COMMAND	STATUS NAME	DESCRIPTION
SXE	**Handled**	The event is considered handled when execution resumes.
SXD,SXN,SXI	**Not Handled**	The event is considered not handled when execution resumes.

To set handling status by using the WinDbg graphical interface, click Event Filters on the **Debug** menu, click the event that you want from the list in the **Event Filters** dialog box, and then select **Handled** or **Not Handled**.

Automatic Commands

The debugger also enables you to set commands that are automatically executed if the event or exception causes a break into the debugger. You can set a command string for the first-chance break and a command string for the second-chance break. You can set these strings with the **SX*** command or the Debug | Event Filters command. Each command string can contain multiple commands that are separated with semicolons.

These commands are executed regardless of the break status. That is, if the break status is "Ignore," the command is still executed. If the break status is "Second-chance break," the first-chance command is executed when the exception first occurs, before any other exception handlers are involved. The command string can end with an execution command such as **g (Go)**, **gh (Go with Exception Handled)**, or **gn (Go with Exception Not Handled)**.

Event Definitions and Defaults

You can change the break status or handling status of the following exceptions. Their default break status is indicated.

The following exceptions' default handling status is always "Not Handled". Be careful about changing this status. If you change this status to "Handled", all first-chance and second-chance exceptions of this type are considered handled, and this configuration bypasses all of the exception-handling routines.

EVENT CODE	MEANING	DEFAULT BREAK STATUS
asrt	Assertion failure	Break
av	Access violation	Break
dm	Data misaligned	Break
dz	Integer division by zero	Break
c000008e	Floating point division by zero	Break
eh	C++ EH exception	Second-chance break
gp	Guard page violation	Break
ii	Illegal instruction	Second-chance break
iov	Integer overflow	Break
ip	In-page I/O error	Break
isc	Invalid system call	Break
lsq	Invalid lock sequence	Break
sbo	Stack buffer overflow	Break
sov	Stack overflow	Break
wkd	Wake debugger	Break
aph	Application hang This exception is triggered if the Windows operating system concludes that a process has stopped responding (that is, *is hung*).	Break
3c	Child application termination	Second-chance break

EVENT CODE	MEANING	DEFAULT BREAK STATUS
chhc	Invalid handle	Break
Number	Any numbered exception	Second-chance break

Note You can override the **asrt** break status for a specific address by using the **ah (Assertion Handling)** command. The **ch** and **hc** event codes refer to the same exception. When you are controlling its break status, use **sx* ch**. When you are controlling its handling status, use **sx* hc**.

You can change the break status or handling status of the following exceptions. Their default break status is indicated.

The following exceptions' default handling status is always "Handled". Because these exceptions are used to communicate with the debugger, you should not typically change their status to "Not Handled". This status causes other exception handlers to catch the exceptions if the debugger ignores them.

An application can use DBG_COMMAND_EXCEPTION (**dbce**) to communicate with the debugger. This exception is similar to a breakpoint, but you can use the **SX*** command to react in a specific way when this exception occurs.

EVENT CODE	MEANING	DEFAULT BREAK STATUS
dbce	Special debugger command exception	Ignore
vcpp	Special Visual C++ exception	Ignore
wos	WOW64 single-step exception	Break
wob	WOW64 breakpoint exception-	Break
sse **ssec**	Single-step exception	Break
bpe **bpec**	Breakpoint exception	Break
cce **cc**	CTRL+C or CTRL+BREAK This exception is triggered if the target is a console application and CTRL+C or CTRL+BREAK is passed to it.	Break

Note The final three exceptions in the preceding table have two different event codes. When you are controlling their break status, use **sse**, **bpe**, and **cce**. When you are controlling their handling status, use **ssec**, **bpec**, and **cc**.

The following exceptions are useful when you are debugging managed code.

EVENT CODE	MEANING	DEFAULT STATUS
clr	Common Language Runtime exception	Second-chance break Not handled
clrn	Common Language Runtime notification exception	Second-chance break Handled

You can change the break status of the following events. Because these events are not exceptions, their handling status is irrelevant.

EVENT CODE	MEANING	DEFAULT BREAK STATUS
ser	System error	Ignore
cpr[**:***Process*]	Process creation Setting the break status of this event applies only to user-mode debugging. This event does not occur in kernel mode. You can control this event only if you have activated debugging of child processes in CDB or WinDbg, either through the -o**command-line option** or through the **.childdbg (Debug Child Processes)** command. The process name can include an optional file name extension and an asterisk (*) or question mark (?) as wildcard characters. The debugger remembers only the most recent ***cpr*** setting. Separate settings for separate processes are not supported. Include a colon or a space between ***cpr*** and Process. *If Process is omitted, the setting applies to any child process creation.*	Ignore

EVENT CODE	MEANING	DEFAULT BREAK STATUS
epr[**:***Process*]	Process exit Setting the break status of this event applies only to user-mode debugging. This event does not occur in kernel mode. You can control this event only if you have activated debugging of child processes in CDB or WinDbg, either through the -o**command-line option** or through the **.childdbg (Debug Child Processes)** command. The process name can include an optional file name extension and an asterisk () or question mark (?) as wildcard characters. The debugger remembers only the most recent **epr** setting. Separate settings for separate processes are not supported. Include a colon or a space between **epr** and *Process*. If *Process* is omitted, the setting applies to any child process exit.	Ignore
ct	Thread creation	Ignore
et	Thread exit	Ignore
ld[**:***Module*]	Load module If you specify *Module*, the break occurs when the module with this name is loaded. *Module* can specify the name or the address of the module. If the name is used, *Module* might contain a variety of wildcard characters and specifiers. (For more information about the syntax, see String Wildcard Syntax.) The debugger remembers only the most recent l**d** setting. Separate settings for separate modules are not supported. Include a colon or a space between **ld** and *Module*. If *Module* is omitted, the event is triggered when any module is loaded.	Output

EVENT CODE	MEANING	DEFAULT BREAK STATUS
ud[**:***Module*]	Unload module If you specify *Module*, the break occurs when the module with this name, or at this base address, is unloaded. *Module* can specify the name or the address of the module. If the name is used, *Module* can be an exact name or include wildcard characters. If *Module* is an exact name, it is immediately resolved to a base address by using the current debugger module list and it is stored as an address. If *Module* contains wildcard characters, the pattern string is kept for later matching when unload events occur. Rarely, the debugger does not have name information for unload events and matches only by the base address. Therefore, if *Module* contains wildcard characters, the debugger cannot perform a name match in this particular unload case and breaks when any module is unloaded. The debugger remembers only the most recent **ud** setting. Separate settings for separate modules are not supported. Include a colon or a space between **ud** and *Module*. If *Module* is omitted, the event is triggered when any module is loaded.	Output
out[**:***Output*]	Target application output If you specify *Output*, the break occurs only when output that matches the specified pattern is received. *Output* can contain a variety of wildcard characters and specifiers. (For more information about the syntax, see String Wildcard Syntax.) However, *Output* cannot contain a colon or spaces. The match is not case sensitive. Include a colon or space between **out** and *Output*.	Ignore

EVENT CODE	MEANING	DEFAULT BREAK STATUS
ibp	Initial break point (This event occurs at the beginning of the debug session and after you restart the target computer.)	**In user mode:** Break. You can change this status to "Ignore" by using the **-g**command-line option. **In kernel mode:** Ignore. You can change this status to "Enabled" by a variety of methods. For more information about how to change this status, see Crashing and Rebooting the Target Computer.
iml	Initial module load (Kernel mode only)	Ignore. You can change this status to "Break" by a variety of methods. For more information about how to change this status, see Crashing and Rebooting the Target Computer.

Finding the process ID

7/18/2019 • 2 minutes to read • Edit Online

Each process running in Microsoft Windows is assigned a unique decimal number called the process ID (PID). This number is used to specify the process when attaching a debugger to it.

You can determine the PID for a given app using Task Manager, the **tasklist** command, the TList utility, or the debugger.

Task Manager

Task Manager can be opened in a number of ways, but the simplest is to select Ctrl+Alt+Delete, and then select **Task Manager**.

On the **Processes** tab, select **Details** to see the PID, along with other useful info.

Some kernel errors may cause delays in Task Manager's graphical interface.

The **tasklist** command

Use the **tasklist** command from a command prompt to display all processes, their PIDs, and a variety of other details.

TList utility

Task List Viewer (TList), or tlist.exe, is a command-line utility that displays the list of tasks, or user-mode processes, currently running on the local computer. TList is included in the Debugging Tools for Windows package.

When you run TList from the command prompt, it will display a list of all the user-mode processes in memory with a unique PID number. For each process, it shows the PID, process name, and, if the process has a window, the title of that window.

For more information, see TList.

The **.tlist** debugger command

If there's already a user-mode debugger running on the system in question, the **.tlist (List Process IDs)** command will display a list of all PIDs on that system.

CSRSS and user-mode drivers

To debug a user-mode driver running on another computer, debug the Client Server Run-Time Subsystem (CSRSS) process. For more information, see Debugging CSRSS.

Debugging a Stack Overflow

1/3/2019 • 6 minutes to read • Edit Online

A stack overflow is an error that user-mode threads can encounter. There are three possible causes for this error:

- A thread uses the entire stack reserved for it. This is often caused by infinite recursion.
- A thread cannot extend the stack because the page file is maxed out, and therefore no additional pages can be committed to extend the stack.
- A thread cannot extend the stack because the system is within the brief period used to extend the page file.

When a function running on a thread allocates local variables, the variables are put on the thread's call stack. The amount of stack space required by the function could be as large as the sum of the sizes of all the local variables. However, the compiler usually performs optimizations that reduce the stack space required by a function. For example, if two variables are in different scopes, the compiler can use the same stack memory for both of those variables. The compiler might also be able to eliminate some local variables entirely by optimizing calculations.

The amount of optimization is influenced by compiler settings applied at build time. For example, a Debug build and a Release build have different levels of optimization. The amount of stack space required by a function in a Debug build might be larger than the amount of stack space required by that same function in a Release build.

Here is an example of how to debug a stack overflow. In this example, NTSD is running on the same computer as the target application and is redirecting its output to KD on the host computer. See Controlling the User-Mode Debugger from the Kernel Debugger for details.

The first step is see what event caused the debugger to break in:

```
0:002> .lastevent
Last event: Exception C00000FD, second chance
```

You can look up exception code 0xC00000FD in ntstatus.h, which can be found in the Microsoft Windows SDK and the Windows Driver Kit (WDK). This exception code is STATUS_STACK_OVERFLOW.

To double-check that the stack overflowed, you can use the **k (Display Stack Backtrace)** command:

```
0:002> k
ChildEBP RetAddr
009fdd0c 71a32520 COMCTL32!_chkstk+0x25
009fde78 77cf8290 COMCTL32!ListView_WndProc+0x4c4
009fde98 77cfd634 USER32!_InternalCallWinProc+0x18
009fdf00 77cd55e9 USER32!UserCallWinProcCheckWow+0x17f
009fdf3c 77cd63b2 USER32!SendMessageWorker+0x4a3
009fdf5c 71a45b30 USER32!SendMessageW+0x44
009fdfec 71a45bb0 COMCTL32!CCSendNotify+0xc0e
009fdffc 71a1d688 COMCTL32!CICustomDrawNotify+0x2a
009fe074 71a1db30 COMCTL32!Header_Draw+0x63
009fe0d0 71a1f196 COMCTL32!Header_OnPaint+0x3f
009fe128 77cf8290 COMCTL32!Header_WndProc+0x4e2
009fe148 77cfd634 USER32!_InternalCallWinProc+0x18
009fe1b0 77cd4490 USER32!UserCallWinProcCheckWow+0x17f
009fe1d8 77cd46c8 USER32!DispatchClientMessage+0x31
009fe200 77f7bb3f USER32!__fnDWORD+0x22
009fe220 77cd445e ntdll!_KiUserCallbackDispatcher+0x13
009fe27c 77cfd634 USER32!DispatchMessageWorker+0x3bc
009fe2e4 009fe4a8 USER32!UserCallWinProcCheckWow+0x17f
00000000 00000000 0x9fe4a8
```

The target thread has broken into COMCTL32!_chkstk, which indicates a stack problem. Now you should investigate the stack usage of the target process. The process has multiple threads, but the important one is the one that caused the overflow, so identify this thread first:

```
0:002> ~*k

   0  id: 570.574   Suspend: 1 Teb 7ffde000 Unfrozen
   .....

   1  id: 570.590   Suspend: 1 Teb 7ffdd000 Unfrozen
   .....

. 2  id: 570.598   Suspend: 1 Teb 7ffdc000 Unfrozen
ChildEBP RetAddr
 009fdd0c 71a32520 COMCTL32!_chkstk+0x25
.....

   3  id: 570.760   Suspend: 1 Teb 7ffdb000 Unfrozen
```

Now you need to investigate thread 2. The period at the left of this line indicates that this is the current thread.

The stack information is contained in the TEB (Thread Environment Block) at 0x7FFDC000. The easiest way to list it is using **!teb**. However, this requires you to have the proper symbols. For maximum versatility, assume you have no symbols:

```
0:002> dd 7ffdc000 L4
7ffdc000   009fdef0 00a00000 009fc000 00000000
```

To interpret this, you need to look up the definition of the TEB data structure. If you had complete symbols, you could use **dt TEB** to do this. But in this case, you will need to look at the ntpsapi.h file in the Microsoft Windows SDK. This file contains the following information:

```
typedef struct _TEB {
    NT_TIB NtTib;
    PVOID  EnvironmentPointer;
    CLIENT_ID ClientId;
    PVOID ActiveRpcHandle;
    PVOID ThreadLocalStoragePointer;
    PPEB ProcessEnvironmentBlock;
    ULONG LastErrorValue;
    .....
    PVOID DeallocationStack;
    .....
} TEB;

typedef struct _NT_TIB {
    struct _EXCEPTION_REGISTRATION_RECORD *ExceptionList;
    PVOID StackBase;
    PVOID StackLimit;
    .....
} NT_TIB;
```

This indicates that the second and third DWORDs in the TEB structure point to the bottom and top of the stack, respectively. In this case, these addresses are 0x00A00000 and 0x009FC000. (The stack grows downward in memory.) You can calculate the stack size using the **? (Evaluate Expression)** command:

```
0:002> ? a00000-9fc000
Evaluate expression: 16384 = 00004000
```

This shows that the stack size is 16 K. The maximum stack size is stored in the field **DeallocationStack**. After some calculation, you can determine that this field's offset is 0xE0C.

```
0:002> dd 7ffdc000+e0c L1
7ffdce0c   009c0000

0:002> ? a00000-9c0000
Evaluate expression: 262144 = 00040000
```

This shows that the maximum stack size is 256 K, which means more than adequate stack space is left.

Furthermore, this process looks clean -- it is not in an infinite recursion or exceeding its stack space by using excessively large stack-based data structures.

Now break into KD and look at the overall system memory usage with the **!vm** extension command:

```
0:002> .breakin
Break instruction exception - code 80000003 (first chance)
ntoskrnl!_DbgBreakPointWithStatus+4:
80148f9c cc               int     3

kd> !vm

*** Virtual Memory Usage ***
        Physical Memory:     16268   (   65072 Kb)
        Page File: \??\C:\pagefile.sys
           Current:    147456Kb Free Space:     65988Kb
           Minimum:     98304Kb Maximum:       196608Kb
        Available Pages:      2299   (    9196 Kb)
        ResAvail Pages:       4579   (   18316 Kb)
        Locked IO Pages:        93   (     372 Kb)
        Free System PTEs:    42754   (  171016 Kb)
        Free NP PTEs:         5402   (   21608 Kb)
        Free Special NP:       348   (    1392 Kb)
        Modified Pages:        757   (    3028 Kb)
        NonPagedPool Usage:    811   (    3244 Kb)
        NonPagedPool Max:     6252   (   25008 Kb)
        PagedPool 0 Usage:    1337   (    5348 Kb)
        PagedPool 1 Usage:     893   (    3572 Kb)
        PagedPool 2 Usage:     362   (    1448 Kb)
        PagedPool Usage:      2592   (   10368 Kb)
        PagedPool Maximum:   13312   (   53248 Kb)
        Shared Commit:        3928   (   15712 Kb)
        Special Pool:         1040   (    4160 Kb)
        Shared Process:       3641   (   14564 Kb)
        PagedPool Commit:     2592   (   10368 Kb)
        Driver Commit:         887   (    3548 Kb)
        Committed pages:     45882   (  183528 Kb)
        Commit limit:        50570   (  202280 Kb)

        Total Private:       33309   (  133236 Kb)
         .....
```

First, look at nonpaged and paged pool usage. Both are well within limits, so these are not the cause of the problem.

Next, look at the number of committed pages: 183528 out of 202280. This is very close to the limit. Although this display does not show this number to be completely at the limit, you should keep in mind that while you are performing user-mode debugging, other processes are running on the system. Each time an NTSD command is executed, these other processes are also allocating and freeing memory. That means you do not know exactly what the memory state was like at the time the stack overflow occurred. Given how close the committed page number is to the limit, it is reasonable to conclude that the page file was used up at some point and this caused the stack overflow.

This is not an uncommon occurrence, and the target application cannot really be faulted for this. If it happens frequently, you may want to consider raising the initial stack commitment for the failing application.

Analyzing a Single Function Call

It can also be useful to find out exactly how much stack space a certain function call is allocating.

To do this, disassemble the first few instructions and look for the instruction `sub esp` *number*. This moves the stack pointer, effectively reserving *number* bytes for local data.

Here is an example:

```
0:002> k
ChildEBP RetAddr
009fdd0c 71a32520 COMCTL32!_chkstk+0x25
009fde78 77cf8290 COMCTL32!ListView_WndProc+0x4c4
009fde98 77cfd634 USER32!_InternalCallWinProc+0x18
009fdf00 77cd55e9 USER32!UserCallWinProcCheckWow+0x17f
009fdf3c 77cd63b2 USER32!SendMessageWorker+0x4a3
009fdf5c 71a45b30 USER32!SendMessageW+0x44
009fdfec 71a45bb0 COMCTL32!CCSendNotify+0xc0e
009fdffc 71a1d688 COMCTL32!CICustomDrawNotify+0x2a
009fe074 71a1db30 COMCTL32!Header_Draw+0x63
009fe0d0 71a1f196 COMCTL32!Header_OnPaint+0x3f
009fe128 77cf8290 COMCTL32!Header_WndProc+0x4e2

0:002> u COMCTL32!Header_Draw
 COMCTL32!Header_Draw :
71a1d625 55              push    ebp
71a1d626 8bec            mov     ebp,esp
71a1d628 83ec58          sub     esp,0x58
71a1d62b 53              push    ebx
71a1d62c 8b5d08          mov     ebx,[ebp+0x8]
71a1d62f 56              push    esi
71a1d630 57              push    edi
71a1d631 33f6            xor     esi,esi
```

This shows that **Header_Draw** allocated 0x58 bytes of stack space.

Manually Walking a Stack

12/5/2018 • 6 minutes to read • Edit Online

In some cases, the stack trace function will fail in the debugger. This can be caused by a call to an invalid address that caused the debugger to lose the location of the return address; or you may have come across a stack pointer for which you cannot directly get a stack trace; or there could be some other debugger problem. In any case, being able to manually walk a stack is often valuable.

The basic concept is fairly simple: dump out the stack pointer, find out where the modules are loaded, find possible function addresses, and verify by checking to see if each possible stack entry makes a call to the next.

Before going through an example, it is important to note that the **kb (Display Stack Backtrace)** command has an additional feature on Intel systems. By doing a kb=[ebp] [eip] [esp], the debugger will display the stack trace for the frame with the given values for base pointer, instruction pointer, and stack pointer, respectively.

For the example, a failure that actually gives a stack trace is used so the results can be checked at the end.

The first step is to find out what modules are loaded where. This is accomplished with the **x (Examine Symbols)** command (some symbols are edited out for reasons of length):

```
kd> x *!
start    end        module name
77f70000 77fb8000   ntdll     (C:\debug\ntdll.dll, \\ntstress\symbols\dll\ntdll.DBG)
80010000 80012320   Aha154x   (load from Aha154x.sys deferred)
80013000 8001aa60   SCSIPORT  (load from SCSIPORT.SYS deferred)
8001b000 8001fba0   Scsidisk  (load from Scsidisk.sys deferred)

80100000 801b7b40   NT        (ntoskrnl.exe, \\ntstress\symbols\exe\ntoskrnl.DBG)
802f0000 8033c000   Ntfs      (load from Ntfs.sys deferred)
80400000 8040c000   hal       (load from hal.dll deferred)
fe4c0000 fe4c38c0   vga       (load from vga.sys deferred)
fe4d0000 fe4d3e60   VIDEOPRT  (load from VIDEOPRT.SYS deferred)
fe4e0000 fe4f0e40   ati       (load from ati.SYS deferred)
fe500000 fe5057a0   Msfs      (load from Msfs.SYS deferred)
fe510000 fe519560   Npfs      (load from Npfs.SYS deferred)

fe520000 fe521f60   ndistapi  (load from ndistapi.sys deferred)
fe530000 fe54ed20   Fastfat   (load from Fastfat.SYS deferred)
fe5603e0 fe575360   NDIS      (NDIS.SYS, \\ntstress\symbols\SYS\NDIS.DBG)
fe580000 fe585920   elnkii    (elnkii.sys, \\ntstress\symbols\sys\elnkii.DBG)
fe590000 fe59b8a0   ndiswan   (load from ndiswan.sys deferred)
fe5a0000 fe5b7c40   nbf       (load from nbf.sys deferred)
fe5c0000 fe5c1b40   TDI       (load from TDI.SYS deferred)
fe5d0000 fe5dd580   nwlnkipx  (load from nwlnkipx.sys deferred)

fe5e0000 fe5ee220   nwlnknb   (load from nwlnknb.sys deferred)
fe5f0000 fe5fb320   afd       (load from afd.sys deferred)
fe610000 fe62bf00   tcpip     (load from tcpip.sys deferred)
fe630000 fe648600   netbt     (load from netbt.sys deferred)
fe650000 fe6572a0   netbios   (load from netbios.sys deferred)
fe660000 fe660000   Parport   (load from Parport.SYS deferred)
fe670000 fe670000   Parallel  (load from Parallel.SYS deferred)
fe680000 fe6bcf20   rdr       (rdr.sys, \\ntstress\symbols\sys\rdr.DBG)

fe6c0000 fe6f0920   srv       (load from srv.sys deferred)
```

The second step is dumping out the stack pointer to look for addresses in the modules given by the **x *!** command:

```
kd> dd esp
fe4cc97c  80136039 00000270 00000000 00000000
fe4cc98c  fe682ae4 801036fe 00000000 fe68f57a
fe4cc99c  fe682a78 ffb5b030 00000000 00000000
fe4cc9ac  ff680e08 801036fe 00000000 00000000
fe4cc9bc  fe6a1198 00000001 fe4cca78 ffae9d98

fe4cc9cc  02000901 fe4cca68 ffb50030 ff680e08
fe4cc9dc  ffa449a8 8011c901 fe4cca78 00000000
fe4cc9ec  80127797 80110008 00000246 fe6a1430

kd> dd
fe4cc9fc  00000270 fe6a10ae 00000270 ffa44abc
fe4cca0c  ffa449a8 ff680e08 fe6b2c04 ff680e08
fe4cca1c  ffa449a8 e12820c8 e1235308 ffa449a8
fe4cca2c  fe685968 ff680e08 e1235308 ffa449a8
fe4cca3c  ffb0ad48 ffb0ad38 00100000 ffb0ad38
fe4cca4c  00000000 ffa44a84 e1235308 0000000a
fe4cca5c  c00000d6 00000000 004ccb28 fe4ccbc4

fe4cca6c  fe680ba4 fe682050 00000000 fe4ccbd4
```

To determine which values are likely function addresses and which are parameters or saved registers, the first thing to consider is what the different types of information look like on the stack. Most integers are going to be smaller value, which means they will be mostly zeros when displayed as DWORDs (like 0x00000270). Most pointers to local addresses will be near the stack pointer (like fe4cca78). Status codes usually begin with a c (c00000d6). Unicode and ASCII strings can be identified by the fact that each character will be in the range of 20-7f. (In KD, the **dc (Display Memory)** command will show the characters on the right.) Most importantly, the function addresses will be in the range listed by **x *!**.

Notice that all modules listed are in the ranges of 77f70000 to 8040c000 and fe4c0000 to fe6f0920. Based on these ranges, the possible function addresses in the preceding list are: 80136039, 801036fe (listed twice, so more likely a parameter), fe682ae4, fe68f57a, fe682a78, fe6a1198, 8011c901, 80127797, 80110008, fe6a1430, fe6a10ae, fe6b2c04, fe685968, fe680ba4, and fe682050. Investigate these locations by using an **ln (List Nearest Symbols)** command for each address:

```
kd> ln 80136039
(80136039)   NT!_KiServiceExit+0x1e  |  (80136039)   NT!_KiServiceExit2-0x177
kd> ln fe682ae4
(fe682ae4)   rdr!_RdrSectionInfo+0x2c | (fe682ae4)   rdr!_RdrFcbReferenceLock-0xb4
kd> ln 801036fe
(801036fe)   NT!_KeWaitForSingleObject | (801036fe)   NT!_MmProbeAndLockPages-0x2f8
kd> ln fe68f57a
(fe68f57a)   rdr!_RdrDereferenceDiscardableCode+0xb4
                       (fe68f57a)   rdr!_RdrUninitializeDiscardableCode-0xa
kd> ln fe682a78
(fe682a78)   rdr!_RdrDiscardableCodeLock | (fe682a78) rdr!_RdrDiscardableCodeTimeout-0x38

kd> ln fe6a1198
(fe6a1198)   rdr!_SubmitTdiRequest+0xae | (fe6a1198)   rdr!_RdrTdiAssociateAddress-0xc
kd> ln 8011c901
(8011c901)   NT!_KeSuspendThread+0x13 | (8011c901)   NT!_FsRtlCheckLockForReadAccess-0x55
kd> ln 80127797
(80127797)   NT!_ZwCloseObjectAuditAlarm+0x7 | (80127797)   NT!_ZwCompleteConnectPort-0x9
kd> ln 80110008
(80110008)   NT!_KeWaitForMultipleObjects+0x27c | (80110008) NT!_FsRtlLookupMcbEntry-0x164
kd> ln fe6a1430
(fe6a1430)   rdr!_RdrTdiCloseConnection+0xa | (fe6a1430)   rdr!_RdrDoTdiConnect-0x4

kd> ln fe6a10ae
(fe6a10ae)   rdr!_RdrTdiDisconnect+0x56 | (fe6a10ae)   rdr!_SubmitTdiRequest-0x3c
kd> ln fe6b2c04
(fe6b2c04)   rdr!_CleanupTransportConnection+0x64 | (fe6b2c04)rdr!_RdrReferenceServer-0x20
kd> ln fe685968
(fe685968)   rdr!_RdrReconnectConnection+0x1b6
                     (fe685968)   rdr!_RdrInvalidateServerConnections-0x32
kd> ln fe682050
(fe682050)   rdr!__strnicmp+0xaa  |  (fe682050)   rdr!_BackPackSpinLock-0xa10
```

As noted before, 801036fe is not likely to be part of the stack trace as it is listed twice. If the return addresses have an offset of zero, they can be ignored (you cannot return to the beginning of a function). Based on this information, the stack trace is revealed to be:

```
NT!_KiServiceExit+0x1e
rdr!_RdrSectionInfo+0x2c
rdr!_RdrDereferenceDiscardableCode+0xb4
rdr!_SubmitTdiRequest+0xae
NT!_KeSuspendThread+0x13
NT!_ZwCloseObjectAuditAlarm+0x7
NT!_KeWaitForMultipleObjects+0x27c
rdr!_RdrTdiCloseConnection+0xa
rdr!_RdrTdiDisconnect+0x56
rdr!_CleanupTransportConnection+0x64
rdr!_RdrReconnectConnection+0x1b6
rdr!__strnicmp+0xaa
```

To verify each symbol, unassemble immediately before the return address specified to see if it does a call to the function above it. To reduce length, the following is edited (the offsets used were found by trial and error):

```
kd> u 80136039-2 l1      //  looks ok, its a call
NT!_KiServiceExit+0x1c:
80136037 ffd3             call    ebx
kd> u fe682ae4-2 l1      //  paged out (all zeroes) unknown
rdr!_RdrSectionInfo+0x2a:
fe682ae2 0000             add     [eax],al
kd> u fe68f57a-6 l1      //  looks ok, its a call, but not anything above
rdr!_RdrDereferenceDiscardableCode+0xae:
fe68f574 ff15203568fe     call dword ptr [rdr!__imp__ExReleaseResourceForThreadLite]
kd> u fe682a78-6 l1      //  paged out (all zeroes) unknown

rdr!_DiscCodeInitialized+0x2:
fe682a72 0000             add     [eax],al
kd> u  fe6a1198-5 l1      //  looks good, call to something above
rdr!_SubmitTdiRequest+0xa9:
fe6a1193 e82ee3feff       call  rdr!_RdrDereferenceDiscardableCode (fe68f4c6)
kd> u 8011c901-2 l1      //  not good, its a jump in the function
NT!_KeSuspendThread+0x11:
8011c8ff 7424             jz      NT!_KeSuspendThread+0x37 (8011c925)
kd> u 80127797-2 l1      //  looks good, an int 2e -> KiServiceExit

NT!_ZwCloseObjectAuditAlarm+0x5:
80127795 cd2e             int     2e
kd> u 80110008-2 l1      //  not good, its a test instruction not a call
NT!_KeWaitForMultipleObjects+0x27a:
80110006 85c9             test    ecx,ecx
kd> u 80110008-5 l1      //  paged out (all zeroes) unknown
NT!_KeWaitForMultipleObjects+0x277:
80110003 0000             add     [eax],al
kd> u fe6a1430-6 l1      //  looks good its a call to ZwClose...
rdr!_RdrTdiCloseConnection+0x4:
fe6a142a ff15f83468fe     call    dword ptr [rdr!__imp__ZwClose (fe6834f8)]

kd> u fe6a10ae-2 l1      //  paged out (all zeroes) unknown
rdr!_RdrTdiDisconnect+0x54:
fe6a10ac 0000             add     [eax],al
kd> u  fe6b2c04-5 l1      //  looks good, call to something above
rdr!_CleanupTransportConnection+0x5f:
fe6b2bff e854e4feff       call    rdr!_RdrTdiDisconnect (fe6a1058)
kd> u fe685968-5 l1      //  looks good, call to immediately above
rdr!_RdrReconnectConnection+0x1b1:
fe685963 e838d20200       call    rdr!_CleanupTransportConnection (fe6b2ba0)

kd> u fe682050-2 l1      //  paged out (all zeroes) unknown
rdr!__strnicmp+0xa8:
fe68204e 0000             add     [eax],al
```

Based on this, it appears that **RdrReconnectConnection** called **RdrCleanupTransportConnection**, to **RdrTdiDisconnect**, to **ZwCloseObjectAuditAlarm**, to **KiSystemServiceExit**. The other functions on the stack are probably leftover portions of previously active stacks.

In this case, the stack trace worked properly. Following is the actual stack trace to check the answer:

```
kd> k
ChildEBP RetAddr
fe4cc978 80136039 NT!_NtClose+0xd
fe4cc978 80127797 NT!_KiServiceExit+0x1e

fe4cc9f4 fe6a1430 NT!_ZwCloseObjectAuditAlarm+0x7
fe4cca10 fe6b2c04 rdr!_RdrTdiCloseConnection+0xa
fe4cca28 fe685968 rdr!_CleanupTransportConnection+0x64
fe4cca78 fe688157 rdr!_RdrReconnectConnection+0x1b6
fe4ccbd4 80106b1e rdr!_RdrFsdCreate+0x45b
fe4ccbe8 8014b289 NT!IofCallDriver+0x38
fe4ccc98 8014decd NT!_IopParseDevice+0x693
fe4ccd08 8014d6d2 NT!_ObpLookupObjectName+0x487
fe4ccde4 8014d3ad NT!_ObOpenObjectByName+0xa2
fe4cce90 8016660d NT!_IoCreateFile+0x433
fe4cced0 80136039 NT!_NtCreateFile+0x2d
```

The first entry was the current location based on the stack trace, but otherwise, the stack was correct up to the point where **RdrReconnectConnection** was called. The same process could have been used to trace the entire stack. For a more exact method of manual stack walking, you would need to unassemble each potential function and follow each **push** and **pop** to identify each DWORD on the stack.

Debugging a Stack Trace that has JScript Frames

12/5/2018 • 2 minutes to read • Edit Online

The JScript Stack Dump Creation and Consumption feature works by collecting JScript frames and stitching them against debugger physical frames. Sometimes on x86 platforms, the debugger constructs the stack trace incorrectly.

If your stack includes JScript frames that you think might be incorrect, enter the following command in the debugger.

.stkwalk_force_frame_pointer 1

Debugging an Application Failure

12/5/2018 • 2 minutes to read • Edit Online

There are a variety of errors possible in user-mode applications.

The most common kinds of failures include access violations, alignment faults, exceptions, critical section time-outs (deadlocks), and in-page I/O errors.

Access violations and data type misalignments are among the most common. They usually occur when an invalid pointer is dereferenced. The blame could lie with the function that caused the fault, or with an earlier function that passed an invalid parameter to the faulting function.

User-mode exceptions have many possible causes. If an unknown exception occurs, locate it in ntstatus.h or winerror.h if possible.

Critical section timeouts (or possible deadlocks) occur when one thread is waiting for a critical section for a long time. These are difficult to debug and require an in-depth analysis of the stack trace.

In-page I/O errors are almost always hardware failures. You can double-check the status code in ntstatus.h to verify.

Reattaching to the Target Application

12/5/2018 • 2 minutes to read • Edit Online

If the debugger freezes or otherwise stops responding (that is, *crashes*) while you perform user-mode debugging, you can attach a new debugger to the existing process.

Note This method is supported only on Microsoft Windows XP and later versions of Windows. This method does not depend on whether the debugger originally created the process or attached to an existing process. This method does not depend on whether you used the **-pd** option.

To reattach a debugger to an existing target application, do the following:

1. Determine the process ID of the target application.
2. Start a new instance of CDB or WinDbg. Use the **-pe** command-line option.

   ```
   Debugger -pe -p PID
   ```

   You can also use other command-line options.

   You can also connect from a dormant debugger by using the **.attach (Attach to Process)** command together with the **-e** option.
3. After the attach is complete, end the original debugger process.
4. If the process does not respond properly, it might have a suspend count that is too high. You can use the **~m (Resume Thread)** command to reduce the suspend count. For more information about suspend counts, see Controlling Processes and Threads.

If the original debugger is still operating properly, this method might not work. The two debuggers are competing for debugging events, and the Windows operating system does not necessarily assign all of the debugging events to the new debugger.

If the original debugger is ended before you attach the new debugger, the target application is also closed. (However, if the debugger attached with the **-pd** option and then exits normally, the target application continues running. In this situation, a second debugger can attach to the target application without using the **-pe** option.)

If you are already debugging a process and want to detach from the process but leave it frozen in a debugging state, you can use the **.abandon (Abandon Process)** command. After this command, any Windows debugger can reattach to the process by using the procedure that is described in this topic.

Crashing and Rebooting the Target Computer

12/5/2018 • 2 minutes to read • Edit Online

When you perform kernel debugging, you can cause the target computer to stop responding (that is, *crash* or *bug check*) by issuing the **.crash (Force System Crash)** command. This command immediately causes the target computer to stop responding. The debugger writes a kernel-mode dump file if you have enabled crash dumps. (For more information about these files, see Creating a Kernel-Mode Dump File.)

To restart the target computer, use the **.reboot (Reboot Target Computer)** command.

If you want the target computer to create a crash dump file and then restart, you should issue the **.crash** command, followed by the **.reboot** command. If you want only to restart, the **.crash** command is not required.

In the early stages of the boot process, the connection between the host computer and the target computer is lost. No information about the target computer is available to the debugger.

After the connection is broken, the debugger closes all symbol files and unloads all debugger extensions. At this point, all breakpoints are lost if you are running KD or CDB. In WinDbg, you can save the current workspace. This action saves all breakpoints.

If you want to end the debugging session at this point, use the **CTRL+B** command (in KD) or click **Exit** on the **File** menu (in WinDbg).

If you do not exit the debugger, the connection is reestablished after enough of the boot process has completed. Symbols and extensions are reloaded at this point. If you are running WinDbg, the kernel-mode workspace is reloaded.

You can tell the debugger to automatically break into the target computer during the restart process at two possible times:

- When the first kernel module is loaded into memory
- When the kernel initializes

To set an automatic breakpoint when the first kernel module loads, use the **-d** command-line option.

You can also change the break state after the debugger is running:

- Control the initial module load and kernel initialization breakpoints like all exceptions and events. You can break into the debugger when these events occur, or ignore them. You can also have a specified command automatically execute when these breakpoints are hit. For more information, see Controlling Exceptions and Events.
- Use the **CTRL+K** shortcut keys in KD, the CTRL+ALT+K shortcut keys in WinDbg, and the Debug | Kernel Connection | Cycle Initial Break command in WinDbg to change the break state. Every time that you use these commands, the debugger switches between three states: no automatic break, break upon kernel initialization, and break on first kernel module load. This method cannot activate both automatic breakpoints at the same time.

Synchronizing with the Target Computer

12/5/2018 • 2 minutes to read • Edit Online

Sometimes during kernel-mode debugging, the target computer stops responding to the debugger.

In KD, you can press **CTRL+R (Re-synchronize)** and then press ENTER to synchronize with the target computer. In WinDbg, use CTRL+ALT+R or Debug | Kernel Connection | Resynchronize.

These commands frequently restore communication between the host and the target. However, resynchronization might not always be successful, especially if you are using a 1394 kernel connection.

The **.restart (Restart Kernel Connection)** command provides a more powerful method of resynchronization. This command is equivalent to exiting the debugger and then attaching a new debugger to the existing session. This command is available only in KD, not in WinDbg.

The **.restart** command is most useful when you are performing remote debugging through remote.exe. (In this kind of debugging, it might be difficult to end and restart the debugger.) However, you cannot use **.restart** from a debugging client if you are performing remote debugging through the debugger.

Finding a Memory Leak

7/15/2019 • 2 minutes to read • Edit Online

A memory leak occurs when a process allocates memory from the paged or nonpaged pools, but does not free the memory. As a result, these limited pools of memory are depleted over time, causing Windows to slow down. If memory is completely depleted, failures may result.

This section includes the following:

- Determining Whether a Leak Exists describes a technique you can use if you are not sure whether there is a memory leak on your system.
- Finding a Kernel-Mode Memory Leak describes how to find a leak that is caused by a kernel-mode driver or component.
- Finding a User-Mode Memory Leak describes how to find a leak that is caused by a user-mode driver or application.

Determining Whether a Leak Exists

12/5/2018 • 2 minutes to read • Edit Online

If Windows performance is degrading over time and you suspect that a memory leak may be involved, the technique described in this section can indicate whether there is a memory leak. It will not tell you what the source of the leak is, nor whether it is user mode or kernel mode.

Begin by launching Performance Monitor. Add the following counters:

- **Memory**-->**Pool Nonpaged Bytes**
- **Memory**-->**Pool Paged Bytes**
- **Paging File**-->**% Usage**

Change the update time to 600 seconds to capture a graph of the leak over time. You might also want to log the data to a file for later examination.

Start the application or test that you believe is causing the leak. Allow the application or test to run undisturbed for some time; do not use the target computer during this time. Leaks are usually slow and may take hours to detect. Wait for a few hours before deciding whether a leak has occurred.

Monitor the Performance Monitor counters. After the test has started, the counter values will change rapidly, and it may take some time for the memory pools values to reach a steady state.

User-mode memory leaks are always located in pageable pool and cause both the **Pool Paged Bytes** counter and the page file **Usage** counter to increase steadily over time. Kernel-mode memory leaks usually deplete nonpaged pool, causing the **Pool Nonpaged Bytes** counter to increase, although pageable memory can be affected as well. Occasionally these counters may show false positives because an application is caching data.

Finding a Kernel-Mode Memory Leak

12/5/2018 • 2 minutes to read • Edit Online

Use the following techniques to determine the cause of a kernel-mode memory leak:

Using PoolMon to Find a Kernel-Mode Memory Leak

Using the Kernel Debugger to Find a Kernel-Mode Memory Leak

Using Driver Verifier to Find a Kernel-Mode Memory Leak

If you do not know which kernel-mode driver or component is responsible for the leak, you should use the PoolMon technique first. This technique reveals the pool tag associated with the memory leak; the driver or component that uses this pool tag is responsible for the leak.

If you have already identified the responsible driver or component, use the second and third techniques in the preceding list to determine the cause of the leak more specifically.

Using PoolMon to Find a Kernel-Mode Memory Leak

12/5/2018 • 2 minutes to read • Edit Online

If you suspect there is a kernel-mode memory leak, the easiest way to determine which pool tag is associated with the leak is to use the PoolMon tool.

PoolMon (Poolmon.exe) monitors pool memory usage by pool tag name. This tool is included in the Windows Driver Kit (WDK). For a full description, see PoolMon in the WDK documentation.

Enable Pool Tagging (Windows 2000 and Windows XP)

On Windows 2000 and Windows XP you must first use GFlags to enable pool tagging. GFlags is included in Debugging Tools for Windows. Start GFlags, choose the **System Registry** tab, check the **Enable Pool Tagging** box, and then click **Apply**. You must restart Windows for this setting to take effect. For more details, see GFlags.

On Windows Server 2003 and later versions of Windows, pool tagging is always enabled.

Using PoolMon

The PoolMon header displays the total paged and non-paged pool bytes. The columns show pool use for each pool tag. The display is updated automatically every few seconds. For example:

```
Memory: 16224K Avail: 4564K PageFlts: 31 InRam Krnl: 684K P: 680K
Commit: 24140K Limit: 24952K Peak: 24932K Pool N: 744K P: 2180K

## Tag   Type  Allocs       Frees        Diff    Bytes       Per Alloc


CM    Paged   1283  ( 0)  1002  ( 0)   281  1377312   ( 0)  4901
Strg  Paged  10385 ( 10)  6658  ( 4)  3727   317952 ( 512)    85
Fat   Paged   6662  ( 8)  4971  ( 6)  1691   174560 ( 128)   103
MmSt  Paged    614  ( 0)   441  ( 0)   173    83456   ( 0)   482
```

PoolMon has command keys that sort the output according to various criteria. Press the letter associated with each command in order to re-sort the data. It takes a few seconds for each command to work.

The sort commands include:

COMMAND KEY	OPERATION
P	Limits the tags shown to nonpaged pool, paged pool, or both. Repeatedly pressing **P** cycles through each of these options, in that order.
B	Sorts tags by maximum byte usage.
M	Sorts tags by maximum byte allocations.
T	Sorts tags alphabetically by tag name.

COMMAND KEY	OPERATION
E	Causes the display to include the paged and non-paged totals across the bottom.
A	Sorts tags by allocation size.
F	Sorts tags by free operations.
S	Sorts tags by the difference between allocations and frees.
Q	Quits PoolMon.

Using the PoolMon Utility to Find a Memory Leak

To find a memory leak with the PoolMon utility, follow this procedure:

1. Start PoolMon.
2. If you have determined that the leak is occurring in non-paged pool, press **P** once; if you have determined that it is occurring in paged pool, press **P** twice. If you do not know, do not press **P** and both kinds of pool are included.
3. Press **B** to sort the display by maximum byte use.
4. Start your test. Take a screen shot and copy it to Notepad.
5. Take a new screen shot every half hour. By comparing screen shots, determine which tag's bytes are increasing.
6. Stop your test and wait a few hours. How much of the tag was freed up in this time?

Typically, after an application reaches a stable running state, it allocates memory and free memory at roughly the same rate. If it tends to allocate memory faster than it frees it, its memory use will grow over time. This often indicates a memory leak.

Addressing the Leak

After you have determined which pool tag is associated with the leak, this might reveal all you need to know about the leak. If you need to determine which specific instance of the allocation routine is causing the leak, see Using the Kernel Debugger to Find Kernel-Mode Memory Leaks.

Using the Kernel Debugger to Find a Kernel-Mode Memory Leak

1/3/2019 • 2 minutes to read • Edit Online

The kernel debugger determines the precise location of a kernel-mode memory leak.

Enable Pool Tagging (Windows 2000 and Windows XP)

You must first use GFlags to enable pool tagging. GFlags is included in Debugging Tools for Windows. Start GFlags, choose the **System Registry** tab, check the **Enable Pool Tagging** box, and then click **Apply**. You must restart Windows for this setting to take effect.

On Windows Server 2003 and later versions of Windows, pool tagging is always enabled.

Determining the Pool Tag of the Leak

To determine which pool tag is associated with the leak, it is usually easiest to use the PoolMon tool for this step. For details, see Using PoolMon to Find Kernel-Mode Memory Leaks.

Alternatively, you can use the kernel debugger to look for tags associated with large pool allocations. To do so, follow this procedure:

1. Reload all modules by using the **.reload (Reload Module)** command.
2. Use the **!poolused** extension. Include the flag "4" to sort the output by paged memory use:

```
kd> !poolused 4
Sorting by Paged Pool Consumed

Pool Used:
            NonPaged            Paged
Tag    Allocs     Used    Allocs     Used
Abc         0        0     36405 33930272
Tron        0        0       552  7863232
IoN7        0        0     10939   998432
Gla5        1      128      2222   924352
Ggb         0        0        22   828384
```

3. Determine which pool tag is associated with the greatest usage of memory. In this example, the driver using the tag "Abc" is using the most memory--almost 34 MB. Therefore, the memory leak is most likely to be in this driver.

Finding the Leak

After you have determined the pool tag associated with the leak, follow this procedure to locate the leak itself:

1. Use the **ed (Enter Values)** command to modify the value of the global system variable **PoolHitTag**. This global variable causes the debugger to break whenever a pool tag matching its value is used.
2. Set **PoolHitTag** equal to the tag that you suspect to be the source of the memory leak. The module name "nt" should be specified for faster symbol resolution. The tag value must be entered in little-endian format (that is, backward). Because pool tags are always four characters, this tag is actually A-b-c-space, not merely A-b-c. So use the following command:

```
kd> ed nt!poolhittag ' cbA'
```

3. To verify the current value of **PoolHitTag**, use the **db (Display Memory)** command:

```
kd> db nt!poolhittag L4
820f2ba4  41 62 63 20           Abc
```

4. The debugger will break every time that pool is allocated or freed with the tag **Abc**. Each time the debugger breaks on one of these allocations or free operations, use the **kb (Display Stack Backtrace)** debugger command to view the stack trace.

Using this procedure, you can determine which code resident in memory is overallocating pool with the tag **Abc**.

To clear the breakpoint, set **PoolHitTag** to zero:

```
kd> ed nt!poolhittag 0
```

If there are several different places where memory with this tag is being allocated and these are in an application or driver that you have written, you can alter your source code to use unique tags for each of these allocations.

If you cannot recompile the program but you want to determine which one of several possible locations in the code is causing the leak, you can unassemble the code at each location and use the debugger to edit this code resident in memory so that each instance uses a distinct (and previously unused) pool tag. Then allow the system to run for several minutes or more. After some time has passed, break in again with the debugger and use the **!poolfind** extension to find all pool allocations associated with each of the new tags.

Using Driver Verifier to Find a Kernel-Mode Memory Leak

12/5/2018 • 2 minutes to read • Edit Online

Driver Verifier determines whether a kernel-mode driver is leaking memory.

The Pool Tracking feature of Driver Verifier monitors the memory allocations made by a specified driver. At the time that the driver is unloaded, Driver Verifier verifies that all allocations made by the driver have been freed. If some of the driver's allocations have not been freed, a bug check is issued, and the parameters of the bug check indicate the nature of the problem.

While this feature is active, use the Driver Verifier Manager graphical interface to monitor pool allocation statistics. If a kernel debugger is attached to the driver, use the **!verifier 0x3** extension to display allocation statistics.

If the driver uses Direct Memory Access (DMA), the DMA Verification feature of Driver Verifier is also helpful in finding memory leaks. DMA Verification tests for a number of common misuses of DMA routines, including failure to free common buffers and other errors that can lead to memory leaks. If a kernel debugger is attached while this option is active, use the **!dma** extension to show allocation statistics.

For information about Driver Verifier, see Driver Verifier in the Windows Driver Kit (WDK) documentation.

Finding a User-Mode Memory Leak

12/5/2018 • 2 minutes to read • Edit Online

Use the following techniques to determine the cause of a user-mode memory leak:

- Using Performance Monitor to Find a User-Mode Memory Leak
- Using UMDH to Find a User-Mode Memory Leak

The first technique determines which process is leaking memory. After you know which process is involved, the second technique can determine the specific routine that is at fault.

Using Performance Monitor to Find a User-Mode Memory Leak

12/5/2018 • 2 minutes to read • Edit Online

If you suspect there is a user-mode memory leak but are not sure which process is causing it, you can use Performance Monitor to measure the memory usage of individual processes.

Launch Performance Monitor. Add the following counters:

- **Process**-->**Private Bytes** (for each process you want to examine)
- **Process**-->**Virtual Bytes** (for each process you wish to examine)

Change the update time to 600 seconds to capture a graph of the leak over time. You might also want to log the data to a file for later examination.

The **Private Bytes** counter indicates the total amount of memory that a process has allocated, not including memory shared with other processes. The **Virtual Bytes** counter indicates the current size of the virtual address space that the process is using.

Some memory leaks appear in the data file as an increase in private bytes allocated. Other memory leaks show up as an increase in the virtual address space.

After you have determined which process is leaking memory, use the UMDH tool to determine the specific routine that is at fault. For details, see Using UMDH to Find User-Mode Memory Leaks.

Using UMDH to Find a User-Mode Memory Leak

1/3/2019 • 5 minutes to read • Edit Online

The user-mode dump heap (UMDH) utility works with the operating system to analyze Windows heap allocations for a specific process. UMDH locates which routine in a specific process is leaking memory.

UMDH is included in Debugging Tools for Windows. For full details, see UMDH.

Preparing to Use UMDH

If you have not already determined which process is leaking memory, do that first. For details, see Using Performance Monitor to Find User-Mode Memory Leaks.

The most important data in the UMDH logs are the stack traces of the heap allocations. To determine whether a process is leaking heap memory, analyze these stack traces.

Before using UMDH to display the stack trace data, you must use GFlags to configure your system properly. GFlags is included in Debugging Tools for Windows.

The following GFlags settings enable UMDH stack traces:

- In the GFlags graphical interface, choose the Image File tab, type the process name (including the file name extension), press the TAB key, select **Create user mode stack trace database**, and then click **Apply**.

  Or, equivalently, use the following GFlags command line, where *ImageName* is the process name (including the file name extension):

  ```
  gflags /i ImageName +ust
  ```

  Use this command to clear the GFlag settings once you are done. For more information, see GFlags Commands.

  ```
  gflags /i ImageName -ust
  ```

- By default, the amount of stack trace data that Windows gathers is limited to 32 MB on an x86 processor, and 64 MB on an x64 processor. If you must increase the size of this database, choose the **Image File** tab in the GFlags graphical interface, type the process name, press the TAB key, check the **Stack Backtrace (Megs)** check box, type a value (in MB) in the associated text box, and then click **Apply**. Increase this database only when necessary, because it may deplete limited Windows resources. When you no longer need the larger size, return this setting to its original value.

- If you changed any flags on the **System Registry** tab, you must restart Windows to make these changes effective. If you changed any flags on the **Image File** tab, you must restart the process to make the changes effective. Changes to the **Kernel Flags** tab are effective immediately, but they are lost the next time Windows restarts.

Before using UMDH, you must have access to the proper symbols for your application. UMDH uses the symbol path specified by the environment variable _NT_SYMBOL_PATH. Set this variable equal to a path containing the symbols for your application. If you also include a path to Windows symbols, the analysis may be more complete. The syntax for this symbol path is the same as that used by the debugger; for details, see Symbol Path.

For example, if the symbols for your application are located at C:\MySymbols, and you want to use the public Microsoft symbol store for your Windows symbols, using C:\MyCache as your downstream store, you would use

the following command to set your symbol path:

```
set _NT_SYMBOL_PATH=c:\mysymbols;srv*c:\mycache*https://msdl.microsoft.com/download/symbols
```

In addition, to assure accurate results, you must disable BSTR caching. To do this, set the OANOCACHE environment variable equal to one (1). Make this setting before you launch the application whose allocations are to be traced.

If you need to trace the allocations made by a service, you must set OANOCACHE as a system environment variable and then restart Windows for this setting to take effect.

Detecting Increases in Heap Allocations with UMDH

After making these preparations, you can use UMDH to capture information about the heap allocations of a process. To do so, follow this procedure:

1. Determine the process ID (PID) for the process you want to investigate.
2. Use UMDH to analyze the heap memory allocations for this process, and save it to a log file. Use the -p switch with the PID, and the -f switch with the name of the log file. For example, if the PID is 124, and you want to name the log file Log1.txt, use the following command:

   ```
   umdh -p:124 -f:log1.txt
   ```dbgcmd
   ```
3. Use Notepad or another program to open the log file. This file contains the call stack for each heap allocation, the number of allocations made through that call stack, and the number of bytes consumed through that call stack.
4. Because you are looking for a memory leak, the contents of a single log file are not sufficient. You must compare log files recorded at different times to determine which allocations are growing.

   UMDH can compare two different log files and display the change in their respective allocation sizes. You can use the greater-than symbol (**>**) to redirect the results into a third text file. You may also want to include the -d option, which converts the byte and allocation counts from hexadecimal to decimal. For example, to compare Log1.txt and Log2.txt, saving the results of the comparison to the file LogCompare.txt, use the following command:

   ```
   umdh log1.txt log2.txt > logcompare.txt
   ```
5. Open the LogCompare.txt file. Its contents resemble the following:

   ```
   + 5320 ( f110 - 9df0) 3a allocs BackTrace00B53
   Total increase == 5320
   ```

   For each call stack (labeled "BackTrace") in the UMDH log files, there is a comparison made between the two log files. In this example, the first log file (Log1.txt) recorded 0x9DF0 bytes allocated for BackTrace00B53, while the second log file recorded 0xF110 bytes, which means that there were 0x5320 additional bytes allocated between the time the two logs were captured. The bytes came from the call stack identified by BackTrace00B53.
6. To determine what is in that backtrace, open one of the original log files (for example, Log2.txt) and search for "BackTrace00B53." The results are similar to this data:
   ```

```
00005320 bytes in 0x14 allocations (@ 0x00000428) by: BackTrace00B53
ntdll!RtlDebugAllocateHeap+0x000000FD
ntdll!RtlAllocateHeapSlowly+0x0000005A
ntdll!RtlAllocateHeap+0x00000808
MyApp!_heap_alloc_base+0x00000069
MyApp!_heap_alloc_dbg+0x000001A2
MyApp!_nh_malloc_dbg+0x00000023
MyApp!_nh_malloc+0x00000016
MyApp!operator new+0x0000000E
MyApp!DisplayMyGraphics+0x0000001E
MyApp!main+0x0000002C
MyApp!mainCRTStartup+0x000000FC
KERNEL32!BaseProcessStart+0x0000003D
```

This UMDH output shows that there were 0x5320 (decimal 21280) total bytes allocated from the call stack. These bytes were allocated from 0x14 (decimal 20) separate allocations of 0x428 (decimal 1064) bytes each.

The call stack is given an identifier of "BackTrace00B53," and the calls in this stack are displayed. In reviewing the call stack, you see that the **DisplayMyGraphics** routine is allocating memory through the **new** operator, which calls the routine *malloc*, which uses the Visual C++ run-time library to obtain memory from the heap.

Determine which of these calls is the last one to explicitly appear in your source code. In this case, it is probably the **new** operator because the call to *malloc* occurred as part of the implementation of **new** rather than as a separate allocation. So this instance of the **new** operator in the **DisplayMyGraphics** routine is repeatedly allocating memory that is not being freed.

Debugging a Time Out

12/5/2018 • 2 minutes to read • Edit Online

There are two main time outs that occur on Windows systems:

Resource Time Outs (kernel mode)

Critical Section Time Outs (user mode)

In many cases, these problems are simply a matter of a thread taking too long to release a resource or exit a section of code.

On a retail system, the time-out value is set high enough that you would not see the break (a true deadlock would simply hang). The time-out values are set in the registry under **HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\SessionManager**. The integer values specify the number of seconds in each time out.

Resource Time Outs

12/5/2018 • 2 minutes to read • Edit Online

During a resource time out, the thread waiting for the resource will break into the kernel debugger with a message similar to the following:

```
Resource @ 800e99c0
 ActiveCount = 0001  Flags = IsOwnedExclusive sharedWaiter
 NumberOfExclusiveWaiters = 0000
  Thread = 809cd2f0, Count = 01
  Thread = 809ebc50, Count = 01
  Thread = 00000000, Count = 00
  Thread = 00000000, Count = 00
  Thread = 00000000, Count = 00
NT!DbgBreakPoint+0x4:
800cee04: 000000ad callkd
```

The thread that holds the lock is the first thread listed (or multiple threads if it is a shared lock). To examine that thread, use a **!thread** extension on the thread ID (809cd2f0, in the previous example). This will give a stack for the thread owning the resource. If it is also waiting for a resource to become available, either the **ExpWaitForResourceExclusive** function or the **ExpWaitForResourceShared** function will be on the stack for that thread.

The first parameter to **ExpWaitForResource*Xxx*** is the lock that is being waited on. To find out about that resource, use a **!locks <*resource id*>** extension, which will give you another thread to check.

If you get to a thread that is not waiting for another resource, that thread is probably the source of the problem. For a list of all held locks, use a **!locks** extension with no parameter at the **kd>** prompt.

Example

```
Resource @ fc664ee0                 // Here's the resource lock address

 ActiveCount = 0001  Flags = IsOwnedExclusive ExclusiveWaiter
 NumberOfExclusiveWaiters = 0001
  Thread = ffaf5410, Count = 01               // Here's the owning thread
  Thread = 00000000, Count = 00
ntoskrnl!_DbgBreakPoint:
80131400 cc               int     3

kd> kb                                       // Start with a stack
ChildEBP RetAddr  Args to Child
fcd44980 801154c0 fc664ee0 ffab45d0 00110001 ntoskrnl!_DbgBreakPoint
fcd4499c 80102521 fc664ee0 ffb08ea8 fcd44a4c ntoskrnl!_ExpWaitForResource+0x114        // Lock being waited
on...

fcd449e8 fc6509fa e12597c8 fef27c08 fee4fca8 ntoskrnl!_ExAcquireResourceExclusiveLite+0xa5
00380020 00000000 00000000 00000000 00000000 nwrdr!_CreateScb+0x2ff

kd> !locks fc664ee0                      // !locks resource address gives lock info
Resource @ nwrdr!_NwScavengerSpinLock (0xfc664ee0)    Exclusively owned
    Contention Count = 45
    NumberOfExclusiveWaiters = 1
     Threads: ffaf5410-01                  // Owning thread again
1 total locks, 1 locks currently held

kd> !thread ffaf5410                     // Check the owning thread

THREAD ffaf5410  Cid e7.e8  Teb: 7ffde000 WAIT: (Executive) KernelMode Non-Alertable
    feecf698  SynchronizationEvent
IRP List:
    fef29208: (0006,00b8) Flags: 00000884  Mdl: feed8328
Not impersonating
Owning Process ffaf5690
WaitTime (seconds)      2781250
Context Switch Count    183175
UserTime                  0:00:23.0153
KernelTime                0:01:01.0187
Start Address 0x77f04644
Initial Sp fec6c000 Current Sp fec6b938
Priority 11 BasePriority 7 PriorityDecrement 0 DecrementCount 8

ChildEBP RetAddr  Args to Child
fec6b950 801044fc feecf668 feecf668 00000080 ntoskrnl!KiSwapContext+0x25
fec6b974 fc655976 feecf698 00000000 00000000 ntoskrnl!_KeWaitForSingleObject+0x218
fec6ba5c fc6509fa e1263968 fef29208 feecf668 nwrdr!_ExchangeWithWait+0x38
fec6ba28 fc6533e5 feecf668 e125b3c8 ffafae08 nwrdr!_CreateScb+0x2ff
fec6bac0 fc652f26 feecf668 fec6bae4 fef29208 nwrdr!_CreateRemoteFile+0x2c9
fec6bb6c fc652b14 feecf668 fef29208 fee50b60 nwrdr!_NwCommonCreate+0x3a2

fec6bbac 80107aea fee50b60 fef29208 804052ac nwrdr!_NwFsdCreate+0x56
fec6bbc0 80142792 fef37700 fec6bdbc fee50b28 ntoskrnl!IofCallDriver+0x38
fec6bd10 80145403 fee50b60 00000000 fec6bdbc ntoskrnl!_IopParseDevice+0x6a0
fec6bd7c 80144c0c 00000000 fec6be34 00000040 ntoskrnl!_ObpLookupObjectName+0x479
fec6be5c 80127803 0012dd64 00000000 80127701 ntoskrnl!_ObOpenObjectByName+0xa2
fec6bef4 801385c3 0012dd64 0012dd3c 00000000 ntoskrnl!_NtQueryAttributesFile+0xc1
fec6bef4 77f716ab 0012dd64 0012dd3c 00000000 ntoskrnl!_KiSystemService+0x83

0012dd20 00000000 00000000 00000000 00000000 ntdll!_ZwQueryAttributesFile+0xb
```

Critical Section Time Outs

12/5/2018 • 3 minutes to read • Edit Online

Critical section time outs can be identified by the stack trace that shows the routine **RtlpWaitForCriticalSection** near the top of the stack. Another variety of critical section time out is a possible deadlock application error. To debug critical section time outs properly, CDB or WinDbg is necessary.

As with resource time outs, the **!ntsdexts.locks** extension will give a list of locks currently held and the threads that own them. Unlike resource time outs, the thread IDs given are not immediately useful. These are system IDs that do not map directly to the thread numbers used by CDB.

Just as with **ExpWaitForResource*Xxx***, *the lock identifier is the first parameter to **RtlpWaitForCriticalSection*. Continue tracing the chain of waits until either a loop is found or the final thread is not waiting for a critical section time out.

Example of Debugging a Critical Time Out

Start by displaying the stack:

```
0:024> kb

ChildEBP RetAddr  Args to Child
0569fca4 77f79c78 77f71000 002a6b88 7fffffff ntdll!_DbgBreakPoint
0569fd04 77f71048 5ffa9f9c 5fef0b4b 5ffa9f9c ntdll!_RtlpWaitForCriticalSection+0x89
0569fd0c 5fef0b4b 5ffa9f9c 002a6b88 002a0019 ntdll!_RtlEnterCriticalSection+0x48
0569fd70 5fedf83f 002a6b88 0569fdc0 0000003e winsrv!_StreamScrollRegion+0x1f0
0569fd8c 5fedfa5b 002a6b88 00190000 00000000 winsrv!_AdjustCursorPosition+0x8e
0569fdc0 5fedf678 0569ff18 0031c200 0335ee88 winsrv!_DoWriteConsole+0x104

0569fefc 5fe6311b 0569ff18 0569ffd0 00000005 winsrv!_SrvWriteConsole+0x96
0569fff4 00000000 00000000 00000024 00000024 csrsrv!_CsrApiRequestThread+0x4ff
```

Now use the **!ntsdexts.locks** extension to find the critical section:

```
0:024> !locks
CritSec winsrv!_ScrollBufferLock at 5ffa9f9c        5ffa9f9c is the first one
LockCount          5
RecursionCount     1
OwningThread       88         // here's the owning thread ID
EntryCount         11c
ContentionCount    135
*** Locked

CritSec winsrv!_gcsUserSrv+0 at 5ffa91b4     //second critical section found below

LockCount          8
RecursionCount     1
OwningThread       6d         // second owning thread
EntryCount         1d6c
ContentionCount    1d47
*** Locked
```

Now search for the thread that has the ID number 0x6D:

```
0:024> ~
  0  id: 16.15   Teb 7ffdd000 Unfrozen
  1  id: 16.13   Teb 7ffdb000 Unfrozen
  2  id: 16.30   Teb 7ffda000 Unfrozen
  3  id: 16.2f   Teb 7ffd9000 Unfrozen
  4  id: 16.2e   Teb 7ffd8000 Unfrozen
  5  id: 16.6c   Teb 7ff6c000 Unfrozen
  6  id: 16.6d   Teb 7ff68000 Unfrozen    // this thread owns the second critical section
  7  id: 16.2d   Teb 7ffd7000 Unfrozen
  8  id: 16.33   Teb 7ffd6000 Unfrozen
  9  id: 16.42   Teb 7ff6f000 Unfrozen
 10  id: 16.6f   Teb 7ff6e000 Unfrozen
 11  id: 16.6e   Teb 7ffd5000 Unfrozen
 12  id: 16.52   Teb 7ff6b000 Unfrozen
 13  id: 16.61   Teb 7ff6a000 Unfrozen
 14  id: 16.7e   Teb 7ff69000 Unfrozen
 15  id: 16.43   Teb 7ff67000 Unfrozen
 16  id: 16.89   Teb 7ff50000 Unfrozen
 17  id: 16.95   Teb 7ff65000 Unfrozen
 18  id: 16.90   Teb 7ff64000 Unfrozen
 19  id: 16.71   Teb 7ff63000 Unfrozen
 20  id: 16.bb   Teb 7ff62000 Unfrozen
 21  id: 16.88   Teb 7ff61000 Unfrozen    // this thread owns the first critical section
 22  id: 16.cd   Teb 7ff5e000 Unfrozen
 23  id: 16.c1   Teb 7ff5f000 Unfrozen
 24  id: 16.bd   Teb 7ff5d000 Unfrozen
```

Thread 21 owns the first critical section. Make that the active thread and get a stack trace:

```
0:024> ~21s
ntdll!_ZwWaitForSingleObject+0xb:
77f71bfb c20c00          ret     0xc

0:021> kb

ChildEBP RetAddr  Args to Child
0556fc44 77f79c20 00000110 00000000 77fa4700 ntdll!_ZwWaitForSingleObject+0xb
0556fcb0 77f71048 5ffa91b4 5feb4f7e 5ffa91b4 ntdll!_RtlpWaitForCriticalSection+0x31
0556fcb8 5feb4f7e 5ffa91b4 0556fd70 77f71000 ntdll!_RtlEnterCriticalSection+0x48
0556fcf4 5fef0b76 01302005 00000000 fffffff4 winsrv!__ScrollDC+0x14
0556fd70 5fedf83f 002bd880 0556fdc0 00000025 winsrv!_StreamScrollRegion+0x21b
0556fd8c 5fedfa5b 002bd880 00190000 00000000 winsrv!_AdjustCursorPosition+0x8e

0556fdc0 5fedf678 0556ff18 002bdf70 002a4d58 winsrv!_DoWriteConsole+0x104
0556fefc 5fe6311b 0556ff18 0556ffd0 00000005 winsrv!_SrvWriteConsole+0x96
0556fff4 00000000 00000000 00000024 00000024 csrsrv!_CsrApiRequestThread+0x4ff
```

Thread 6 owns the second critical section. Examine its stack as well:

```
0:021> ~6s
winsrv!_PtiFromThreadId+0xd:
5fe8429a 394858          cmp     [eax+0x58],ecx    ds:0023:7f504da8=000000f8

0:006> kb

ChildEBP RetAddr  Args to Child
01ecfeb4 5fecd0d7 00000086 00000000 7f5738e0 winsrv!_PtiFromThreadId+0xd
01ecfed0 5feccf62 00000086 01ecfff4 00000113 winsrv!__GetThreadDesktop+0x12
01ecfefc 5fe6311b 01ecff18 01ecffd0 00000005 winsrv!___GetThreadDesktop+0x8b
01ecfff4 00000000 00000000 00000024 00000024 csrsrv!_CsrApiRequestThread+0x4ff
```

Thread 21 has **RtlpWaitForCriticalSection** near the top of its stack. Thread 6 does not. So thread 21 is the culprit.

Debugging a Stalled System

12/5/2018 • 2 minutes to read • Edit Online

There are times when the computer can stop responding without actually initiating a bug check. This "freeze" can appear in a variety of forms:

- The mouse pointer can be moved, but does not affect any windows on the screen.
- The entire screen is still and the mouse pointer does not move, but paging continues between the memory and the disk.
- The screen is still and the disk is silent.

If the mouse pointer moves or there is paging to the disk, this is usually due to a problem within the Client Server Run-Time Subsystem (CSRSS).

If NTSD is running on CSRSS, press F12 and dump out each thread to see if there is anything out of the ordinary. (See Debugging CSRSS for more details.)

If an examination of CSRSS reveals nothing, then the problem may be with the kernel after all.

If there is no mouse movement or paging, then it is almost certainly a kernel problem.

Analyzing a kernel crash of this sort is generally a difficult task. To begin, break into KD (with **CTRL+C**) or WinDbg (with **CTRL+BREAK**). You can now use the debugger commands to examine the situation.

Some useful techniques in this case include:

Finding the Failed Process

Debugging an Interrupt Storm

Finding the Failed Process

12/5/2018 • 2 minutes to read • Edit Online

Before finding the failed process, make sure that you are in the context of the accepting processor. To determine the accepting processor, use the **!pcr** extension on each processor and looking for the processor for which an exception handler has been loaded. The exception handler of the accepting processor has an address other than 0xFFFFFFFF.

For example, because the address of **NtTib.ExceptionList** on this processor, is 0xFFFFFFFF, this is not the processor with the failed process:

```
0: kd> !pcr
PCR Processor 0 @ffdff000
 NtTib.ExceptionList: ffffffff
            NtTib.StackBase: 80470650
           NtTib.StackLimit: 8046d860
         NtTib.SubSystemTib: 00000000
              NtTib.Version: 00000000
          NtTib.UserPointer: 00000000
              NtTib.SelfTib: 00000000

                    SelfPcr: ffdff000
                       Prcb: ffdff120
                       Irql: 00000000
                        IRR: 00000000
                        IDR: ffffffff
              InterruptMode: 00000000
                        IDT: 80036400
                        GDT: 80036000
                        TSS: 80257000

              CurrentThread: 8046c610
                 NextThread: 00000000
                 IdleThread: 8046c610

                  DpcQueue:
```

However, the results for Processor 1 are quite different. In this case, the value of **NtTib.ExceptionList** is **f0823cc0**, not 0xFFFFFFFF, indicating that this is the processor on which the exception occurred.

```
0: kd> ~1
1: kd> !pcr
PCR Processor 1 @81497000
 NtTib.ExceptionList: f0823cc0
             NtTib.StackBase: f0823df0
            NtTib.StackLimit: f0821000
          NtTib.SubSystemTib: 00000000
               NtTib.Version: 00000000
           NtTib.UserPointer: 00000000
               NtTib.SelfTib: 00000000

                     SelfPcr: 81497000
                        Prcb: 81497120
                        Irql: 00000000
 IRR: 00000000
                         IDR: ffffffff
               InterruptMode: 00000000
                         IDT: 8149b0e8
 GDT: 8149b908
                         TSS: 81498000

               CurrentThread: 81496d28
                  NextThread: 00000000
                  IdleThread: 81496d28

                   DpcQueue:
```

When you are in the correct processor context, the **!process** extension displays the currently running process.

The most interesting parts of the process dump are:

- The times (a high value indicates that process might be the culprit).
- The handle count (this is the number in parentheses after **ObjectTable** in the first entry).
- The thread status (many processes have multiple threads). If the current process is *Idle*, it is likely that either the machine is truly idle or it hung due to some unusual problem.

Although using the **!process 0 7** extension is the best way to find the problem on a hung system, it is sometimes too much information to filter. Instead, use a **!process 0 0** and then a **!process** on the process handle for CSRSS and any other suspicious processes.

When using a **!process 0 7**, many of the threads might be marked "kernel stack not resident" because those stacks are paged out. If those pages are still in the cache that is in transition, you can get more information by using a **.cache decodeptes** before **!process 0 7**:

```
kd> .cache decodeptes
kd> !process 0 7
```

If you can identify the failing process, use **!process** <*process*> **7** to show the kernel stacks for each thread in the process. This output can identify the problem in kernel mode and reveal what the suspect process is calling.

In addition to **!process**, the following extensions can help to determine the cause of an unresponsive computer:

EXTENSION	EFFECT
!ready	Identifies the threads that are ready to run, in order of priority.

EXTENSION	EFFECT
!kdext*.locks	Identifies any held resource locks, in case there is a deadlock with retail time outs.
!vm	Checks the virtual memory usage.
!poolused	Determines whether one type of pool allocation is disproportionately large (pool tagging required).
!memusage	Checks the physical memory status.
!heap	Checks the validity of the heap.
!irpfind	Searches nonpaged pool for active IRPs.

If the information provided does not indicate an unusual condition, try setting a breakpoint at **ntoskrnl!KiSwapThread** to determine whether the processor is stuck in one process or if it is still scheduling other processes. If it is not stuck, set breakpoints in common functions, such as **NtReadFile**, to determine whether the computer is stuck in a specific code path.

Debugging an Interrupt Storm

12/5/2018 • 7 minutes to read • Edit Online

One of the most common examples of a stalled system is an interrupt storm. An *interrupt storm* is a level-triggered interrupt signal that remains in the asserted state.

The following events can cause an interrupt storm:

- A hardware device does not release its interrupt signal after being directed to do so by the device driver.
- A device driver does not instruct its hardware to release the interrupt signal, because it does not detect that the interrupt was initiated from its hardware.
- A device driver claims the interrupt even though the interrupt was not initiated from its hardware. This situation can only occur when multiple devices are sharing the same IRQ.
- The edge level control register (ELCR) is not set correctly.
- Edge and level interrupt-triggered devices share an IRQ (for example, a COM port and a PCI SCSI controller).

This example demonstrates one method for detecting and debugging an interrupt storm.

When the machine hangs, use a kernel debugger to break in. Use the **!irpfind** extension command to look for pending IRPs. Then, use the **!irp** extension to obtain details about any pending IRPs. For example:

```
kd> !irp 81183468
Irp is active with 2 stacks 2 is current (= 0x811834fc)
 No Mdl Thread 00000000:  Irp stack trace.
     cmd  flg cl Device   File     Completion-Context
 [  0, 0]   0  0 8145f470 00000000 00000000-00000000
               \Driver\E100B
                        Args: 00000000 00000000 00000000 00000000
>[ 16, 2]   0 e1 8145f470 00000000 8047f744-814187a8 Success Error Cancel pending
               \Driver\E100B    ntoskrnl!PopCompleteSystemPowerIrp
                        Args: 00000000 00000000 00000002 00000002
```

This example shows that \driver\e100b has not returned the IRP for **ntoskrnl!PopCompleteSystemPowerIrp**. It appears to be stuck and might be experiencing an interrupt storm.

To investigate, use the **kb** command to request a stack trace. For example:

```
kd> kb
ChildEBP RetAddr  Args to Child
f714ee68 8046355a 00000001 80068c10 00000030 ntoskrnl!RtlpBreakWithStatusInstruction
f714ee68 80067a4f 00000001 80068c10 00000030 ntoskrnl!KeUpdateSystemTime+0x13e
f714eeec 8046380b 01001010 0000003b f714ef00 halacpi!HalBeginSystemInterrupt+0x83
f714eeec 80463c50 01001010 0000003b f714ef00 ntoskrnl!KiChainedDispatch+0x1b
f714ef78 80067cc2 00000000 00000240 8000017c ntoskrnl!KiDispatchInterrupt
f714ef78 80501cb5 00000000 00000240 8000017c halacpi!HalpDispatchInterrupt2ndEnt
```

Notice that the section in bold is an interrupt dispatch. If you use the **g** command and break in again, you will very likely see a different stack trace, but you will still see an interrupt dispatch. To determine which interrupt is responsible for the system stall, look at the second parameter passed into **HalBeginSystemInterrupt** (in this case, 0x3B). The standard rule is that the interrupt vector displayed (0x3B) is the IRQ line plus 0x30, so the

interrupt is number 0xB. Running another stack trace may provide more information about which device issued the interrupt service request (ISR). In this case, a second stack trace has the following result:

```
kd> kb
ChildEBP RetAddr  Args to Child
f714ee24 8046355a 00000001 00000010 00000030 ntoskrnl!RtlpBreakWithStatusInstruction
f714ee24 bfe854b9 00000001 00000010 00000030 ntoskrnl!KeUpdateSystemTime+0x13e
f714eed8 f7051796 00000000 80463850 8143ec88 atimpab!AtiInterrupt+0x109
f714eee0 80463850 8143ec88 81444038 8046380b VIDEOPRT!pVideoPortInterrupt+0x16
f714eef8 80463818 00000202 0000003b 80450bb8 ntoskrnl!KiChainedDispatch2ndLvl+0x28
f714eef8 80463c50 00000202 0000003b 80450bb8 ntoskrnl!KiChainedDispatch+0x28
f714ef78 80067cc2 00000000 00000240 8000017c ntoskrnl!KiDispatchInterrupt
f714ef78 80501cb5 00000000 00000240 8000017c halacpi!HalpDispatchInterrupt2ndEntry+0x1b
f714f084 8045f744 f714f16c 00020019 f714f148 ntoskrnl!NtCreateKey+0x113
f714f084 8042e487 f714f16c 00020019 f714f148 ntoskrnl!KiSystemService+0xc4
f714f118 804ab556 f714f16c 00020019 f714f148 ntoskrnl!ZwCreateKey+0xb
f714f184 8041f75b f714f1e8 8000017c f714f1d0 ntoskrnl!IopCreateRegistryKeyEx+0x4e
f714f204 804965cd 8145f630 00000000 00000001 ntoskrnl!IopProcessSetInterfaceState+0x93
f714f220 bfee1eb9 8145f630 00000000 8145f5a0 ntoskrnl!IoSetDeviceInterfaceState+0x2b
f714f254 bfedb416 00000004 00000800 0045f570 NDIS!ndisMCommonHaltMiniport+0x1f
f714f268 bfed4ddb bfed0660 811a2708 811a2708 NDIS!ndisPmHaltMiniport+0x9a
f714f288 bfed5146 811a2708 00000004 8145f570 NDIS!ndisSetPower+0x1d1
f714f2a8 8041c60f 81453a30 811a2708 80475b18 NDIS!ndisPowerDispatch+0x84
f714f2bc 8044cc52 80475b18 811a2708 811a279c ntoskrnl!IopfCallDriver+0x35
f714f2d4 8044cb89 811a279c 811a2708 811a27c0 ntoskrnl!PopPresentIrp+0x62
```

The system is currently running the ISR for the video card. The system will run the ISR for each of the devices sharing IRQ 0xB. If no process claims the interrupt, the operating system will wait infinitely, requesting the driver ISRs to handle the interrupt. It is also possible that a process might handle the interrupt and stop it, but if the hardware is broken the interrupt may simply be re-asserted.

Use the **!arbiter 4** extension to determine which devices are on IRQ 0xB. If there is only one device on IRQ 0xB, you have found the cause of the problem.. If there is more than one device sharing the interrupt (99% of the cases), you will need to isolate the device either by manually programming LNK nodes (which is destructive to the system state), or by removing or disabling hardware.

```
kd> !arbiter 4
DEVNODE 8149a008 (HTREE\ROOT\0)
  Interrupt Arbiter "RootIRQ" at 80472a20
    Allocated ranges:
      0000000000000000 - 0000000000000000   B   8149acd0
      0000000000000001 - 0000000000000001   B   8149acd0
      0000000000000002 - 0000000000000002   B   8149acd0
      0000000000000003 - 0000000000000003   B   8149acd0
      0000000000000004 - 0000000000000004   B   8149acd0
      0000000000000005 - 0000000000000005   B   8149acd0
      0000000000000006 - 0000000000000006   B   8149acd0
      0000000000000007 - 0000000000000007   B   8149acd0
      0000000000000008 - 0000000000000008   B   8149acd0
      0000000000000009 - 0000000000000009   B   8149acd0
      000000000000000a - 000000000000000a   B   8149acd0
      000000000000000b - 000000000000000b   B   8149acd0
      000000000000000c - 000000000000000c   B   8149acd0
 000000000000000d - 000000000000000d   B   8149acd0
      000000000000000e - 000000000000000e   B   8149acd0
 000000000000000f - 000000000000000f   B   8149acd0
      0000000000000010 - 0000000000000010   B   8149acd0
      0000000000000011 - 0000000000000011   B   8149acd0
      0000000000000012 - 0000000000000012   B   8149acd0
      0000000000000013 - 0000000000000013   B   8149acd0
      0000000000000014 - 0000000000000014   B   8149acd0
      0000000000000015 - 0000000000000015   B   8149acd0
      0000000000000016 - 0000000000000016   B   8149acd0
      0000000000000017 - 0000000000000017   B   8149acd0
```

```
      0000000000000018 - 0000000000000018   B   8149acd0
      0000000000000019 - 0000000000000019   B   8149acd0
      000000000000001a - 000000000000001a   B   8149acd0
      000000000000001b - 000000000000001b   B   8149acd0
      000000000000001c - 000000000000001c   B   8149acd0
      000000000000001d - 000000000000001d   B   8149acd0
 000000000000001e - 000000000000001e   B   8149acd0
      000000000000001f - 000000000000001f   B   8149acd0
 0000000000000020 - 0000000000000020   B   8149acd0
      0000000000000021 - 0000000000000021   B   8149acd0
      0000000000000022 - 0000000000000022   B   8149acd0
      0000000000000023 - 0000000000000023   B   8149acd0
      0000000000000024 - 0000000000000024   B   8149acd0
      0000000000000025 - 0000000000000025   B   8149acd0
      0000000000000026 - 0000000000000026   B   8149acd0
      0000000000000027 - 0000000000000027   B   8149acd0
      0000000000000028 - 0000000000000028   B   8149acd0
      0000000000000029 - 0000000000000029   B   8149acd0
      000000000000002a - 000000000000002a   B   8149acd0
      000000000000002b - 000000000000002b   B   8149acd0
      000000000000002c - 000000000000002c   B   8149acd0
 000000000000002d - 000000000000002d   B   8149acd0
      000000000000002e - 000000000000002e   B   8149acd0
 000000000000002f - 000000000000002f   B   8149acd0
      0000000000000032 - 0000000000000032   B   8149acd0
      0000000000000039 - 0000000000000039 S     814776d0  (ACPI)
   Possible allocation:
     < none >

   DEVNODE 81476f28 (ACPI_HAL\PNP0C08\0)
     Interrupt Arbiter "ACPI_IRQ" at bfff10e0
       Allocated ranges:
         0000000000000000 - 0000000000000000   B   81495bb0
         0000000000000001 - 0000000000000001       814952b0  (i8042prt)
         0000000000000003 - 0000000000000003 S     81495610  (Serial)
         0000000000000004 - 0000000000000004   B   8149acd0
         0000000000000006 - 0000000000000006       81495730  (fdc)
         0000000000000008 - 0000000000000008       81495a90
         0000000000000009 - 0000000000000009 S     814776d0  (ACPI)
         000000000000000b - 000000000000000b S
           000000000000000b - 000000000000000b S     81453c30  (ds1)
           000000000000000b - 000000000000000b S     81453a30  (E100B)
           000000000000000b - 000000000000000b S     81493c30  (uhcd)
           000000000000000b - 000000000000000b S     8145c390  (atirage3)
         000000000000000c - 000000000000000c       814953d0  (i8042prt)
 000000000000000d - 000000000000000d   B   81495850
         000000000000000e - 000000000000000e       8145bb50  (atapi)
 000000000000000f - 000000000000000f       8145b970  (atapi)
       Possible allocation:
         < none >
```

In this case, the audio, Universal Serial Bus (USB), network interface card (NIC), and video are all using the same IRQ.

To find out which ISR claims ownership of the interrupt, examine the return value from the ISR. Simply disassemble the ISR using the **U** command with address given in the **!arbiter** display, and set a breakpoint on the last instruction of the ISR (which will be a 'ret' instruction). Note that using the command **g <address>** is the equivalent of setting a breakpoint on that address:

```
kd> g bfe33e7b
ds1wdm!AdapterIsr+ad:
bfe33e7b c20800          ret     0x8
```

Use the **r** command to examine the registers. In particular, look at the EAX register. If the portion of the register

contents in bold (in the following code example) is anything other then zero, this ISR claimed the interrupt. Otherwise, the interrupt was not claimed, and the operating system will call the next ISR. This example shows that the video card is not claiming the interrupt:

```
kd> r
eax=00000000 ebx=813f4ff0 ecx=00000010 edx=ffdff848 esi=8145d168 edi=813f4fc8
eip=bfe33e7b esp=f714eec4 ebp=f714eee0 iopl=0         nv up ei pl zr na po nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00000246
ds1wdm!AdapterIsr+ad:
bfe33e7b c20800          ret     0x8
```

In fact, in this case, the interrupt is not claimed by any of the devices on IRQ 0xb. When you encounter this problem, you should also check to see if each piece of hardware associated with the interrupt is actually enabled. For PCI, this is easy -- look at the CMD register displayed by the **!pci** extension output:

```
kd> !pci 0 0
PCI Bus 0
00:0  8086:7190.03  Cmd[0006:.mb...]  Sts[2210:c....]  Device  Host bridge
01:0  8086:7191.03  Cmd[0107:imb..s]  Sts[0220:.6...]  PciBridge 0->1-1  PCI-PCI bridge
03:0  1073:000c.03  Cmd[0000:......]  Sts[0210:c....]  Device  SubID:1073:000c Audio device
04:0  8086:1229.05  Cmd[0007:imb...]  Sts[0290:c....]  Device  SubID:8086:0008 Ethernet
07:0  8086:7110.02  Cmd[000f:imb...]  Sts[0280:.....]  Device  ISA bridge
07:1  8086:7111.01  Cmd[0005:i.b...]  Sts[0280:.....]  Device  IDE controller
07:2  8086:7112.01  Cmd[0005:i.b...]  Sts[0280:.....]  Device  USB host controller
07:3  8086:7113.02  Cmd[0003:im....]  Sts[0280:.....]  Device  Class:6:80:0
```

Note that the audio chip's CMD register is zero. This means the audio chip is effectively disabled at this time. This also means that the audio chip will not be capable of responding to accesses by the driver.

In this case, the audio chip needs to be manually re-enabled.

Debugging Multiple Targets

12/5/2018 • 4 minutes to read • Edit Online

You can debug multiple dump files or live user-mode applications at the same time. Each target contains one or more processes, and each process contains one or more threads.

These targets are also grouped into *systems*. Systems are sets of targets that are grouped together for easy identification and manipulation. Systems are defined as follows:

- Each kernel-mode or user-mode dump file is a separate system.
- When you are debugging live user-mode applications on different computers (by using a process server, such as Dbgsrv), each application is a separate system.
- When you are debugging live user-mode applications on the local computer, the applications are combined into a single system.

The *current* or *active* system is the system that you are currently debugging.

Acquiring Multiple Targets

The first target is acquired in the usual manner.

You can debug additional live user-mode applications by using the **.attach (Attach to Process)** or **.create (Create Process)** command, followed by the **g (Go)** command.

You can debug additional dump files by using the **.opendump (Open Dump File)** command, followed by the **g (Go)** command. You can also open multiple dump files when the debugger is started. To open multiple dump files, include multiple **-z** switches in the command, each followed by a different file name.

You can use the preceding commands even if the processes are on different systems. You must start a process server on each system and then use the -premote parameter with **.attach** or **.create** to identify the proper process server. If you use the **.attach** or **.create** command again without specifying the -premote parameter, the debugger attaches to, or creates, a process on the current system.

Manipulating Systems and Targets

When debugging begins, the current system is the one that the debugger most recently attached to. If an exception occurs, the current system switches to the system that this exception occurred on.

To close one target and continue to debug the other targets, use the **.kill (Kill Process)** command. You can use the **.detach (Detach from Process)** command or WinDbg's Debug | Detach Debuggee menu command instead. These commands detach the debugger from the target but leave the target running.

To control the debugging of multiple systems, you can use the following methods:

- The **|| (System Status)** command displays information about one or more systems
- The **||s (Set Current System)** command enables you to select the current system
- (WinDbg only) The Processes and Threads window enables you display or select systems, processes, and threads

By using these commands to select the current system, and by using the standard commands to select the current process and thread, you can determine the context of commands that display memory and registers.

However, you cannot separate execution of these processes. The **g (Go)** command always causes all targets to

execute together.

Note There are complications, when you debug live targets and dump targets together, because commands behave differently for each type of debugging. For example, if you use the **g (Go)** command when the current system is a dump file, the debugger begins executing, but you cannot break back into the debugger, because the break command is not recognized as valid for dump file debugging.

Example

To work with three dump files at the same time, you can use the -z option to load them when WinDbg is started.

```
windbg -z c:\notepad.dmp -z c:\paint.dmp -z c:\calc.dmp
```

For more infomation see WinDbg Command-Line Options. You can also use the .opendump and the **g (Go)** commands to load additional dump files in the debugger.

Use the || (System Status) command to confirm that all three systems are present.

```
||0:0:007> ||
.  0 User mini dump: c:\notepad.dmp
   1 User mini dump: C:\paint.dmp
   2 User mini dump: c:\calc.dmp
```

Use the **g (Go)** command to complete loading of the dump files.

```
||0:0:007> g

************* Path validation summary **************
Response                         Time (ms)     Location
Deferred                                       srv*
Symbol search path is: srv*
Executable search path is:
Windows 10 Version 15063 MP (4 procs) Free x64
Product: WinNt, suite: SingleUserTS
15063.0.amd64fre.rs2_release.170317-1834
Machine Name:
Debug session time: Fri Jun  9 15:52:04.000 2017 (UTC - 7:00)
System Uptime: not available
Process Uptime: 0 days 0:03:44.000
.....................................................
This dump file has a breakpoint exception stored in it.
The stored exception information can be accessed via .ecxr.
ntdll!DbgBreakPoint:
00007ff8`aada8d70 cc              int     3
```

Then use the ||s (Set Current System) command to set the current system to system 1 and then display the current system.

```
||1:1:017> ||1s
||1:1:017> ||
   0 User mini dump: c:\notepad.dmp
.  1 User mini dump: c:\paint.dmp
   2 User mini dump: c:\calc.dmp
```

You can use the .detach command when you are done looking at the current dump file.

```
||1:1:017> .detach
ntdll!DbgBreakPoint:
00007ff8`aada8d70 cc              int     3
Detached
||0:0:007> ||
.  0 User mini dump: c:\notepad.dmp
   2 User mini dump: c:\calc.dmp
```

Resources

For addtional information on debugging see the following resources.

Books

- Advanced Windows Debugging by Mario Hewardt and Daniel Pravat
- Inside Windows Debugging: A Practical Guide to Debugging and Tracing Strategies in Windows by Tarik Soulami
- Windows Internals by Pavel Yosifovich, Alex Ionescu, Mark E. Russinovich and David A. Solomon

Video

The Defrag Tools Show WinDbg Episodes 13-29 https://channel9.msdn.com/Shows/Defrag-Tools

Tracking Down a Processor Hog

12/5/2018 • 3 minutes to read • Edit Online

If one application is consuming ("hogging") all the processor's attention, other processes will end up "starving" and unable to run.

Use the following procedure to correct a bug of this sort.

Debugging an application that is using all the CPU cycles

1. **Identify which application is causing this problem:** Use **Task Manager** or **Perfmon** to find which process is using 99% or 100% of the processor's cycles. This may tell you the offending thread as well.
2. Attach WinDbg, KD, or CDB to this process.
3. **Identify which thread is causing the problem:** Break into the offending application. Use the **!runaway 3** extension to take a "snapshot" of where all the CPU time is going. Use **g (Go)** and wait a few seconds. Then break in and use **!runaway 3** again.

```
0:002> !runaway 3
 User Mode Time
 Thread    Time
 4e0        0:12:16.0312
 268        0:00:00.0000
 22c        0:00:00.0000
 Kernel Mode Time
 Thread    Time
 4e0        0:00:05.0312
 268        0:00:00.0000
 22c        0:00:00.0000

0:002> g

0:001> !runaway 3
 User Mode Time
 Thread    Time
 4e0        0:12:37.0609
 3d4        0:00:00.0000
 22c        0:00:00.0000
 Kernel Mode Time
 Thread    Time
 4e0        0:00:07.0421
 3d4        0:00:00.0000
 22c        0:00:00.0000
```

   Compare the two sets of numbers and look for the thread whose user-mode time or kernel-mode time has increased the most. Because **!runaway** sorts by descending CPU time, the offending thread is usually the one at the top of the list. In this case, thread 0x4E0 is causing the problem.

4. Use the **~ (Thread Status)** and **~s (Set Current Thread)** commands to make this the current thread:

```
0:001> ~
   0  Id: 3f4.3d4 Suspend: 1 Teb: 7ffde000 Unfrozen
.  1  Id: 3f4.22c Suspend: 1 Teb: 7ffdd000 Unfrozen
 2  Id: 3f4.4e0 Suspend: 1 Teb: 7ffdc000 Unfrozen

0:001> ~2s
```

5. Use **kb (Display Stack Backtrace)** to obtain a stack trace of this thread:

```
0:002> kb
FramePtr  RetAddr   Param1   Param2   Param3   Function Name
0b4ffc74  77f6c600  000000c8.00000000 77fa5ad0 BuggyProgram!CreateMsgFile+0x1b
0b4ffce4  01836060  0184f440 00000001 0b4ffe20 BuggyProgram!OpenDestFileStream+0xb3
0b4ffd20  01843eba  02b5b920 00000102 02b1e0e0 BuggyProgram!SaveMsgToDestFolder+0xb3
0b4ffe20  01855924  0b4ffef0 00145970 0b4ffef0 BuggyProgram!DispatchToConn+0xa4
0b4ffe5c  77e112e6  01843e16 0b4ffef0 0b4fff34 RPCRT4!DispatchToStubInC+0x34
0b4ffeb0  77e11215  0b4ffef0 00000000 0b4fff34 RPCRT4!?
DispatchToStubWorker@RPC_INTERFACE@@AAEJPAU_RPC_MESSAGE@@IPAJ@Z+0xb0
0b4ffed0  77e1a3b1  0b4ffef0 00000000 0b4fff34 RPCRT4!?
DispatchToStub@RPC_INTERFACE@@QAEJPAU_RPC_MESSAGE@Z+0x41
0b4fff40  77e181e4  02b1e0b0 00000074 0b4fff90 RPCRT4!?ReceiveOriginalCall@OSF_SCONNECTION@Z+0x14b
0b4fff60  77e1a5df  02b1e0b0 00000074 00149210 RPCRT4!?DispatchPacket@OSF_SCONNECTION@+0x91
0b4fff90  77e1ac1c  77e15eaf 00149210 0b4fffec RPCRT4!?ReceiveLotsaCalls@OSF_ADDRESS@@QAEXXZ+0x76
```

6. Set a breakpoint on the return address of the currently-running function. In this case, the return address is shown on the first line as 0x77F6C600. The return address is equivalent to the function offset shown on the second line (**BuggyProgram!OpenDestFileStream+0xB3**). If no symbols are available for the application, the function name may not appear. Use the **g (Go)** command to execute until this return address is reached, using either the symbolic or hexadecimal address:

```
0:002> g BuggyProgram!OpenDestFileStream+0xb3
```

7. If this breakpoint is hit, repeat the process. For example, suppose this breakpoint is hit. The following steps should be taken:

```
0:002> kb
FramePtr  RetAddr   Param1   Param2   Param3   Function Name
0b4ffce4  01836060  0184f440 00000001 0b4ffe20 BuggyProgram!OpenDestFileStream+0xb3
0b4ffd20  01843eba  02b5b920 00000102 02b1e0e0 BuggyProgram!SaveMsgToDestFolder+0xb3
0b4ffe20  01855924  0b4ffef0 00145970 0b4ffef0 BuggyProgram!DispatchToConn+0xa4
0b4ffe5c  77e112e6  01843e16 0b4ffef0 0b4fff34 RPCRT4!DispatchToStubInC+0x34
0b4ffeb0  77e11215  0b4ffef0 00000000 0b4fff34 RPCRT4!?
DispatchToStubWorker@RPC_INTERFACE@@AAEJPAU_RPC_MESSAGE@@IPAJ@Z+0xb0
0b4ffed0  77e1a3b1  0b4ffef0 00000000 0b4fff34 RPCRT4!?
DispatchToStub@RPC_INTERFACE@@QAEJPAU_RPC_MESSAGE@Z+0x41
0b4fff40  77e181e4  02b1e0b0 00000074 0b4fff90 RPCRT4!?ReceiveOriginalCall@OSF_SCONNECTION@Z+0x14b
0b4fff60  77e1a5df  02b1e0b0 00000074 00149210 RPCRT4!?DispatchPacket@OSF_SCONNECTION@+0x91
0b4fff90  77e1ac1c  77e15eaf 00149210 0b4fffec RPCRT4!?ReceiveLotsaCalls@OSF_ADDRESS@@QAEXXZ+0x76

0:002> g BuggyProgram!SaveMsgToDestFolder+0xb3
```

If this is hit, continue with:

```
0:002> kb
FramePtr  RetAddr   Param1   Param2   Param3   Function Name
0b4ffd20  01843eba  02b5b920 00000102 02b1e0e0 BuggyProgram!SaveMsgToDestFolder+0xb3
0b4ffe20  01855924  0b4ffef0 00145970 0b4ffef0 BuggyProgram!DispatchToConn+0xa4
0b4ffe5c  77e112e6  01843e16 0b4ffef0 0b4fff34 RPCRT4!DispatchToStubInC+0x34
0b4ffeb0  77e11215  0b4ffef0 00000000 0b4fff34 RPCRT4!?
DispatchToStubWorker@RPC_INTERFACE@@AAEJPAU_RPC_MESSAGE@@IPAJ@Z+0xb0
0b4ffed0  77e1a3b1  0b4ffef0 00000000 0b4fff34 RPCRT4!?
DispatchToStub@RPC_INTERFACE@@QAEJPAU_RPC_MESSAGE@Z+0x41
0b4fff40  77e181e4  02b1e0b0 00000074 0b4fff90 RPCRT4!?ReceiveOriginalCall@OSF_SCONNECTION@Z+0x14b
0b4fff60  77e1a5df  02b1e0b0 00000074 00149210 RPCRT4!?DispatchPacket@OSF_SCONNECTION@+0x91
0b4fff90  77e1ac1c  77e15eaf 00149210 0b4fffec RPCRT4!?ReceiveLotsaCalls@OSF_ADDRESS@@QAEXXZ+0x76

0:002> g BuggyProgram!DispatchToConn+0xa4
```

8. Finally you will find a breakpoint that is not hit. In this case, you should assume that the last **g** command set the target running and it did not break. This means that the **SaveMsgToDestFolder()** function will never return.

9. Break into the thread again and set a breakpoint on **BuggyProgram!SaveMsgToDestFolder+0xB3** with the **bp (Set Breakpoint)** command. Then use the **g** command repeatedly. If this breakpoint hits immediately, regardless of how many times you have executed the target, it is very likely that you have identified the offending function:

```
0:002> bp BuggyProgram!SaveMsgToDestFolder+0xb3

0:002> g

0:002> g
```

10. Use the **p (Step)** command to proceed through the function until you identify the place where the looping sequence of instructions are. You can then analyze the application's source code to identify the cause of the spinning thread. The cause will usually turn out to be a problem in the logic of a **while**, **do-while**, **goto**, or **for** loop.

Determining the ACL of an Object

12/5/2018 • 2 minutes to read • Edit Online

You can use the debugger to examine the access control list (ACL) of an object.

The following method can be used if you are performing kernel debugging. To use it while you are performing user-mode debugging, you need to redirect control to a kernel debugger. See Controlling the User-Mode Debugger from the Kernel Debugger for details.

First, use the **!object** debugger extension with the name of the object in question:

```
kd> !object \BaseNamedObjects\AgentToWkssvcEvent
Object: ffbb8a98  Type: (80e30e70) Event
    ObjectHeader: ffbb8a80
    HandleCount: 2  PointerCount: 3
    Directory Object: e14824a0  Name: AgentToWkssvcEvent
```

This shows that the object header has address 0xFFBB8A80. Use the **dt (Display Type)** command with this address and the **nt!_OBJECT_HEADER** structure name:

```
kd> dt nt!_OBJECT_HEADER ffbb8a80
   +0x000 PointerCount     : 3
   +0x004 HandleCount      : 2
   +0x004 NextToFree       : 0x00000002
 +0x008 Type             : 0x80e30e70
   +0x00c NameInfoOffset   : 0x10 '
 +0x00d HandleInfoOffset : 0 '
   +0x00e QuotaInfoOffset  : 0 '
   +0x00f Flags            : 0x20 ' '
   +0x010 ObjectCreateInfo : 0x8016b460
   +0x010 QuotaBlockCharged : 0x8016b460
   +0x014 SecurityDescriptor : 0xe11f08b6
   +0x018 Body             : _QUAD
```

The security descriptor pointer value is shown as 0xE11F08B6. The lowest 3 bits of this value represent an offset past the beginning of this structure, so you should ignore them. In other words, the SECURITY_DESCRIPTOR structure actually begins at 0xE11F08B6 & ~0x7. Use the **!sd** extension on this address:

```
kd> !sd e11f08b0
->Revision: 0x1
->Sbz1    : 0x0
->Control : 0x8004
            SE_DACL_PRESENT
 SE_SELF_RELATIVE
->Owner   : S-1-5-32-544
->Group   : S-1-5-18
->Dacl    :
->Dacl    : ->AclRevision: 0x2
->Dacl    : ->Sbz1       : 0x0
->Dacl    : ->AclSize    : 0x44
->Dacl    : ->AceCount   : 0x2
->Dacl    : ->Sbz2       : 0x0
->Dacl    : ->Ace[0]: ->AceType: ACCESS_ALLOWED_ACE_TYPE
->Dacl    : ->Ace[0]: ->AceFlags: 0x0
->Dacl    : ->Ace[0]: ->AceSize: 0x14
->Dacl    : ->Ace[0]: ->Mask : 0x001f0003
->Dacl    : ->Ace[0]: ->SID: S-1-5-18

->Dacl    : ->Ace[1]: ->AceType: ACCESS_ALLOWED_ACE_TYPE
->Dacl    : ->Ace[1]: ->AceFlags: 0x0
->Dacl    : ->Ace[1]: ->AceSize: 0x18
->Dacl    : ->Ace[1]: ->Mask : 0x00120001
->Dacl    : ->Ace[1]: ->SID: S-1-5-32-544

->Sacl    :  is NULL
```

This displays the security information for this object.

Displaying a Critical Section

8/26/2019 • 5 minutes to read • Edit Online

Critical sections can be displayed in user mode by a variety of different methods. The exact meaning of each field depends on the version of Microsoft Windows version you are using.

Displaying Critical Sections

Critical sections can be displayed by the **!ntsdexts.locks** extension, the **!critsec** extension, the **!cs** extension, and the **dt (Display Type)** command.

The **!ntsdexts.locks** extension displays a list of critical sections associated with the current process. If the **-v** option is used, all critical sections are displayed. Here is an example:

```
0:000> !locks

CritSec ntdll!FastPebLock+0 at 77FC49E0
LockCount          0
RecursionCount     1
OwningThread       c78
EntryCount         0
ContentionCount    0
*** Locked

....
Scanned 37 critical sections
```

If you know the address of the critical section you wish to display, you can use the **!critsec** extension. This displays the same collection of information as **!ntsdexts.locks**. For example:

```
0:000> !critsec 77fc49e0

CritSec ntdll!FastPebLock+0 at 77FC49E0
LockCount          0
RecursionCount     1
OwningThread       c78
EntryCount         0
ContentionCount    0
*** Locked
```

The **!cs** extension can display a critical section based on its address, search an address range for critical sections, and even display the stack trace associated with each critical section. Some of these features require full Windows symbols to work properly. If Application Verifier is active, **!cs -t** can be used to display the critical section tree. See the **!cs** reference page for details and examples.

The information displayed by **!cs** is slightly different than that shown by **!ntsdexts.locks** and **!critsec**. For example:

```
## 0:000> !cs 77fc49e0

Critical section   = 0x77fc49e0 (ntdll!FastPebLock+0x0)
DebugInfo          = 0x77fc3e00
LOCKED
LockCount          = 0x0
OwningThread       = 0x00000c78
RecursionCount     = 0x1
LockSemaphore      = 0x0
SpinCount          = 0x00000000
```

The **dt (Display Type)** command can be used to display the literal contents of the RTL_CRITICAL_SECTION structure. For example:

```
0:000> dt RTL_CRITICAL_SECTION 77fc49e0
   +0x000 DebugInfo        : 0x77fc3e00
   +0x004 LockCount        : 0
   +0x008 RecursionCount   : 1
   +0x00c OwningThread     : 0x00000c78
   +0x010 LockSemaphore    : (null)
   +0x014 SpinCount        : 0
```

Interpreting Critical Section Fields in Windows XP and Windows 2000

The most important fields of the critical section structure are as follows:

- In Microsoft Windows 2000, and Windows XP, the **LockCount** field indicates the number of times that any thread has called the **EnterCriticalSection** routine for this critical section, minus one. This field starts at -1 for an unlocked critical section. Each call of **EnterCriticalSection** increments this value; each call of **LeaveCriticalSection** decrements it. For example, if **LockCount** is 5, this critical section is locked, one thread has acquired it, and five additional threads are waiting for this lock.
- The **RecursionCount** field indicates the number of times that the owning thread has called **EnterCriticalSection** for this critical section.
- The **EntryCount** field indicates the number of times that a thread other than the owning thread has called **EnterCriticalSection** for this critical section.

A newly initialized critical section looks like this:

```
0:000> !critsec 433e60
CritSec mymodule!cs+0 at 00433E60
LockCount          NOT LOCKED
RecursionCount     0
OwningThread       0
EntryCount         0
ContentionCount    0
```

The debugger displays "NOT LOCKED" as the value for **LockCount**. The actual value of this field for an unlocked critical section is -1. You can verify this with the **dt (Display Type)** command:

```
0:000> dt RTL_CRITICAL_SECTION 433e60
   +0x000 DebugInfo        : 0x77fcec80
   +0x004 LockCount        : -1
   +0x008 RecursionCount   : 0
   +0x00c OwningThread     : (null)
   +0x010 LockSemaphore    : (null)
   +0x014 SpinCount        : 0
```

When the first thread calls the **EnterCriticalSection** routine, the critical section's **LockCount**, **RecursionCount**, **EntryCount** and **ContentionCount** fields are all incremented by one, and **OwningThread** becomes the thread ID of the caller. **EntryCount** and **ContentionCount** are never decremented. For example:

```
0:000> !critsec 433e60
CritSec mymodule!cs+0 at 00433E60
LockCount          0
RecursionCount     1
OwningThread       4d0
EntryCount         0
ContentionCount    0
```

At this point, four different things can happen.

1. The owning thread calls **EnterCriticalSection** again. This will increment **LockCount** and **RecursionCount**. **EntryCount** is not incremented.

   ```
   0:000> !critsec 433e60
   CritSec mymodule!cs+0 at 00433E60
   LockCount          1
   RecursionCount     2
   OwningThread       4d0
   EntryCount         0
   ContentionCount    0
   ```

2. A different thread calls **EnterCriticalSection**. This will increment **LockCount** and **EntryCount**. **RecursionCount** is not incremented.

   ```
   0:000> !critsec 433e60
   CritSec mymodule!cs+0 at 00433E60
   LockCount          1
   RecursionCount     1
   OwningThread       4d0
   EntryCount         1
   ContentionCount    1
   ```

3. The owning thread calls **LeaveCriticalSection**. This will decrement **LockCount** (to -1) and **RecursionCount** (to 0), and will reset **OwningThread** to 0.

   ```
   0:000> !critsec 433e60
   CritSec mymodule!cs+0 at 00433E60
   LockCount          NOT LOCKED
   RecursionCount     0
   OwningThread       0
   EntryCount         0
   ContentionCount    0
   ```

4. Another thread calls **LeaveCriticalSection**. This produces the same results as the owning thread calling **LeaveCriticalSection** -- it will decrement **LockCount** (to -1) and **RecursionCount** (to 0), and will reset **OwningThread** to 0.

When any thread calls **LeaveCriticalSection**, Windows decrements **LockCount** and **RecursionCount**. This feature has both good and bad aspects. It allows a device driver to enter a critical section on one thread and leave the critical section on another thread. However, it also makes it possible to accidentally call **LeaveCriticalSection** on the wrong thread, or to call **LeaveCriticalSection** too many times and cause **LockCount** to reach values lower than -1. This corrupts the critical section and causes all threads to wait indefinitely on the critical section.

Interpreting Critical Section Fields in Windows Server 2003 SP1 and Later

In Microsoft Windows Server 2003 Service Pack 1 and later versions of Windows, the **LockCount** field is parsed as follows:

- The lowest bit shows the lock status. If this bit is 0, the critical section is locked; if it is 1, the critical section is not locked.
- The next bit shows whether a thread has been woken for this lock. If this bit is 0, then a thread has been woken for this lock; if it is 1, no thread has been woken.
- The remaining bits are the ones-complement of the number of threads waiting for the lock.

As an example, suppose the **LockCount** is -22. The lowest bit can be determined in this way:

```
0:009> ? 0x1 & (-0n22)
Evaluate expression: 0 = 00000000
```

The next-lowest bit can be determined in this way:

```
0:009> ? (0x2 & (-0n22)) >> 1
Evaluate expression: 1 = 00000001
```

The ones-complement of the remaining bits can be determined in this way:

```
0:009> ? ((-1) - (-0n22)) >> 2
Evaluate expression: 5 = 00000005
```

In this example, the first bit is 0 and therefore the critical section is locked. The second bit is 1, and so no thread has been woken for this lock. The complement of the remaining bits is 5, and so there are five threads waiting for this lock.

Additional Information

For information about how to debug critical section time outs, see Critical Section Time Outs. For general information about critical sections, see the Microsoft Windows SDK, the Windows Driver Kit (WDK), or *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Debugging a Deadlock

6/10/2019 • 6 minutes to read • Edit Online

When a thread needs exclusive access to code or some other resource, it requests a *lock*. If it can, Windows responds by giving this lock to the thread. At this point, nothing else in the system can access the locked code. This happens all the time and is a normal part of any well-written multithreaded application. Although a particular code segment can only have one lock on it at a time, multiple code segments can each have their own lock.

A *deadlock* arises when two or more threads have requested locks on two or more resources, in an incompatible sequence. For instance, suppose that Thread One has acquired a lock on Resource A and then requests access to Resource B. Meanwhile, Thread Two has acquired a lock on Resource B and then requests access to Resource A. Neither thread can proceed until the other thread's lock is relinquished, and, therefore, neither thread can proceed.

User-mode deadlocks arise when multiple threads, usually of a single application, have blocked each other's access to the same resource. However, multiple threads of multiple applications can also block each other's access to a global/shared resource, such as a global event, or semaphore.

Kernel-mode deadlocks arise when multiple threads (from the same process or from distinct processes) have blocked each others' access to the same kernel resource.

The procedure used to debug a deadlock depends on whether the deadlock occurs in user mode or in kernel mode.

Debugging a User-Mode Deadlock

When a deadlock occurs in user mode, use the following procedure to debug it:

1. Issue the **!ntsdexts.locks** extension. In user mode, you can just type **!locks** at the debugger prompt; the **ntsdexts** prefix is assumed.
2. This extension displays all the critical sections associated with the current process, along with the ID for the owning thread and the lock count for each critical section. If a critical section has a lock count of zero, it is not locked. Use the **~ (Thread Status)** command to see information about the threads that own the other critical sections.
3. Use the **kb (Display Stack Backtrace)** command for each of these threads to determine whether they are waiting on other critical sections.
4. Using the output of these **kb** commands, you can find the deadlock: two threads that are each waiting on a lock held by the other thread. In rare cases, a deadlock could be caused by more than two threads holding locks in a circular pattern, but most deadlocks involve only two threads.

Here is an illustration of this procedure. You begin with the **!ntdexts.locks** extension:

```
0:006>  !locks
CritSec ftpsvc2!g_csServiceEntryLock+0 at 6833dd68
LockCount          0
RecursionCount     1
OwningThread       a7
EntryCount         0
ContentionCount    0
*** Locked

CritSec isatq!AtqActiveContextList+a8 at 68629100
LockCount          2
RecursionCount     1
OwningThread       a3
EntryCount         2
ContentionCount    2
*** Locked

CritSec +24e750 at 24e750
LockCount          6
RecursionCount     1
OwningThread       a9
EntryCount         6
ContentionCount    6
*** Locked
```

The first critical section displayed has no locks and, therefore, can be ignored.

The second critical section displayed has a lock count of 2 and is, therefore, a possible cause of a deadlock. The owning thread has a thread ID of 0xA3.

You can find this thread by listing all threads with the **~ (Thread Status)** command, and looking for the thread with this ID:

```
0:006>  ~
   0  Id: 1364.1330 Suspend: 1 Teb: 7ffdf000 Unfrozen
   1  Id: 1364.17e0 Suspend: 1 Teb: 7ffde000 Unfrozen
   2  Id: 1364.135c Suspend: 1 Teb: 7ffdd000 Unfrozen
   3  Id: 1364.1790 Suspend: 1 Teb: 7ffdc000 Unfrozen
   4  Id: 1364.a3 Suspend: 1 Teb: 7ffdb000 Unfrozen
   5  Id: 1364.1278 Suspend: 1 Teb: 7ffda000 Unfrozen
.  6  Id: 1364.a9 Suspend: 1 Teb: 7ffd9000 Unfrozen
   7  Id: 1364.111c Suspend: 1 Teb: 7ffd8000 Unfrozen
   8  Id: 1364.1588 Suspend: 1 Teb: 7ffd7000 Unfrozen
```

In this display, the first item is the debugger's internal thread number. The second item (the `Id` field) contains two hexadecimal numbers separated by a decimal point. The number before the decimal point is the process ID; the number after the decimal point is the thread ID. In this example, you see that thread ID 0xA3 corresponds to thread number 4.

You then use the **kb (Display Stack Backtrace)** command to display the stack that corresponds to thread number 4:

```
0:006>  ~4 kb
  4  id: 97.a3   Suspend: 0 Teb 7ffd9000 Unfrozen
ChildEBP RetAddr  Args to Child
014cfe64 77f6cc7b 00000460 00000000 00000000 ntdll!NtWaitForSingleObject+0xb
014cfed8 77f67456 0024e750 6833adb8 0024e750 ntdll!RtlpWaitForCriticalSection+0xaa
014cfee0 6833adb8 0024e750 80000000 01f21cb8 ntdll!RtlEnterCriticalSection+0x46
014cfef4 6833ad8f 01f21cb8 000a41f0 014cff20 ftpsvc2!DereferenceUserDataAndKill+0x24
014cff04 6833324a 01f21cb8 00000000 00000079 ftpsvc2!ProcessUserAsyncIoCompletion+0x2a
014cff20 68627260 01f21e0c 00000000 00000079 ftpsvc2!ProcessAtqCompletion+0x32
014cff40 686249a5 000a41f0 00000001 686290e8 isatq!I_TimeOutContext+0x87
014cff5c 68621ea7 00000000 00000001 0000001e isatq!AtqProcessTimeoutOfRequests_33+0x4f
014cff70 68621e66 68629148 000ad1b8 686230c0 isatq!I_AtqTimeOutWorker+0x30
014cff7c 686230c0 00000000 00000001 000c000a isatq!I_AtqTimeoutCompletion+0x38
014cffb8 77f04f2c 00000000 00000001 000c000a isatq!SchedulerThread_297+0x2f
00000001 000003e6 00000000 00000001 000c000a kernel32!BaseThreadStart+0x51
```

Notice that this thread has a call to the **WaitForCriticalSection** function, which means that not only does it have a lock, it is waiting for code that is locked by something else. We can find out which critical section we are waiting on by looking at the first parameter of the call to **WaitForCriticalSection**. This is the first address under **Args to Child**: "24e750". So this thread is waiting on the critical section at address 0x24E750. This was the third critical section listed by the **!locks** extension that you used earlier.

In other words, thread 4, which owns the second critical section, is waiting on the third critical section. Now turn your attention to the third critical section, which is also locked. The owning thread has thread ID 0xA9. Returning to the output of the **~** command that you saw previously, note that the thread with this ID is thread number 6. Display the stack backtrace for this thread:

```
0:006>  ~6 kb
ChildEBP RetAddr  Args to Child
0155fe38 77f6cc7b 00000414 00000000 00000000 ntdll!NtWaitForSingleObject+0xb
0155feac 77f67456 68629100 6862142e 68629100 ntdll!RtlpWaitForCriticalSection+0xaa
0155feb4 6862142e 68629100 0009f238 686222e1 ntdll!RtlEnterCriticalSection+0x46
0155fec0 686222e1 0009f25c 00000001 0009f238 isatq!ATQ_CONTEXT_LISTHEAD__RemoveFromList
0155fed0 68621412 0009f238 686213d1 0009f238 isatq!ATQ_CONTEXT__CleanupAndRelease+0x30
0155fed8 686213d1 0009f238 00000001 01f26bcc isatq!AtqpReuseOrFreeContext+0x3f
0155fee8 683331f7 0009f238 00000001 01f26bf0 isatq!AtqFreeContext+0x36
0155fefc 6833984b ffffffff 00000000 00000000 ftpsvc2!ASYNC_IO_CONNECTION__SetNewSocket
0155ff18 6833adcd 77f05154 01f26a58 00000000 ftpsvc2!USER_DATA__Cleanup+0x47
0155ff28 6833ad8f 01f26a58 000a3410 0155ff54 ftpsvc2!DereferenceUserDataAndKill+0x39
0155ff38 6833324a 01f26a58 00000000 00000040 ftpsvc2!ProcessUserAsyncIoCompletion+0x2a
0155ff54 686211eb 01f26bac 00000000 00000040 ftpsvc2!ProcessAtqCompletion+0x32
0155ff88 68622676 000a3464 00000000 000a3414 isatq!AtqpProcessContext+0xa7
0155ffb8 77f04f2c abcdef01 ffffffff 000ad1b0 isatq!AtqPoolThread+0x32
0155ffec 00000000 68622644 abcdef01 00000000 kernel32!BaseThreadStart+0x51
```

This thread, too, is waiting for a critical section to be freed. In this case, it is waiting on the critical section at 0x68629100. This was the second critical section in the list generated earlier by the **!locks** extension.

This is the deadlock. Thread 4, which owns the second critical section, is waiting on the third critical section. Thread 6, which owns the third critical section, is waiting on the second critical section.

Having confirmed the nature of this deadlock, you can use the usual debugging techniques to analyze threads 4 and 6.

Debugging a Kernel-Mode Deadlock

There are several debugger extensions that are useful for debugging deadlocks in kernel mode:

- The **!kdexts.locks** extension displays information about all locks held on kernel resources and the threads holding these locks. (In kernel mode, you can just type **!locks** at the debugger prompt; the **kdexts** prefix is assumed.)

- The **!qlocks** extension displays the state of all queued spin locks.
- The **!wdfkd.wdfspinlock** extension displays information about a Kernel-Mode Driver Framework (KMDF) spin-lock object.
- The **!deadlock** extension is used in conjunction with Driver Verifier to detect inconsistent use of locks in your code that have the potential to cause deadlocks.

When a deadlock occurs in kernel mode, use the **!kdexts.locks** extension to list all the locks currently acquired by threads.

You can usually pinpoint the deadlock by finding one non-executing thread that holds an exclusive lock on a resource that is required by an executing thread. Most of the locks are shared.

Debugging a Failed Driver Unload

12/5/2018 • 2 minutes to read • Edit Online

A driver will not unload if there is a leaked reference to **DeviceObject** or **DriverObject**. This is a common cause of failed driver unloads.

Apart from **IoCreateDevice**, there are several functions that take reference to **DriverObject** and **DeviceObject**. If you do not follow the guidelines for using the functions, you will end up leaking the reference.

Here is an example of how to debug this problem. Although **DeviceObject** is used in this example, this technique works for all objects.

Fixing a driver that fails to unload

1. Put a breakpoint right after the driver calls **IoCreateDevice**. Get the **DeviceObject** address.
2. Find the object header by using the **!object** extension on this object address:

   ```
   kd> !object 81a578c0
   Object: 81a578c0  Type: (81bd0e70) Device
       ObjectHeader: 81a578a8
       HandleCount: 0  PointerCount: 3
       Directory Object: e1001208  Name: Serial0
   ```

   The first variable in the **ObjectHeader** is the *pointer count* or *reference count*.
3. Put a write breakpoint on the pointer count, using the **ObjectHeader**'s address:

   ```
   kd> ba w4 81a578a8 "k;g"
   ```
4. Use **g (Go)**. The debugger will produce a log.
5. Look for the mismatched reference/dereference pair -- specifically, a missing dereference. (Note that **ObReferenceObject** is implemented as a macro inside the kernel.)

Reading Bug Check Callback Data

10/23/2019 • 3 minutes to read • Edit Online

Many drivers supply *bug check callback routines*. When Windows issues a bug check, it calls these routines before shutting down the system. These routines can specify and write to areas of memory known as *callback data* and *secondary callback data*.

BugCheckCallback
Data written by this routine becomes part of callback data. The data is not included in the crash dump file.

BugCheckSecondaryDumpDataCallback
Data written by this routine becomes part of secondary callback data. The data is included in the crash dump file.

BugCheckAddPagesCallback
Pages specified by this routine become part of callback data. The data in those pages is included in the crash dump file.

The amount of callback and secondary callback data that is available to the debugger depends on several factors:

- If you are performing live debugging of a crashed system, callback data that has already been written by BugCheckCallback or specified by BugCheckAddPagesCallback will be available. Secondary callback data will not be available, because it is not stored in any fixed memory location.
- If you are debugging a Complete Memory Dump or Kernel Memory Dump, callback data specified by BugCheckAddPagesCallback and secondary callback data written by BugCheckSecondaryDumpDataCallback will be available. Callback data written by BugCheckCallback will not be available.
- If you are debugging a Small Memory Dump, callback data will not be available. Secondary callback data will be available.

See Varieties of Kernel-Mode Dump Files for more details on these different dump file sizes.

Displaying Callback Data

To display bug check callback data, you can use the **!bugdump** extension.

Without any parameters, **!bugdump** will display data for all callbacks.

To view data for one specific callback routine, use **!bugdump***Component*, where *Component* is the same parameter that was passed to **KeRegisterBugCheckCallback** when that routine was registered.

Displaying Secondary Callback Data

There are two methods for displaying secondary callback data. You can use the **.enumtag** command or you can write your own debugger extension.

Each block of secondary callback data is identified by a GUID tag. This tag is specified by the **Guid** field of the **(KBUGCHECK_SECONDARY_DUMP_DATA)ReasonSpecificData** parameter passed to BugCheckSecondaryDumpDataCallback.

The **.enumtag (Enumerate Secondary Callback Data)** command is not a very precise instrument. It displays every secondary data block, showing the tag and then showing the data in hexadecimal and ASCII format. It is generally useful only to determine what tags are actually being used for secondary data blocks.

To use this data in a more practical way, it is recommended that you write your own debugger extension. This

extension must call methods in the dbgeng.h header file. For details, see Writing New Debugger Extensions.

If you know the GUID tag of the secondary data block, your extension should use the method **IDebugDataSpaces3::ReadTagged** to access the data. Its prototype is as follows:

```
STDMETHOD(ReadTagged)(
    THIS_
    IN LPGUID Tag,
    IN ULONG Offset,
    OUT OPTIONAL PVOID Buffer,
    IN ULONG BufferSize,
    OUT OPTIONAL PULONG TotalSize
    ) PURE;
```

Here is an example of how to use this method:

```
UCHAR RawData[MY_DATA_SIZE];
GUID MyGuid = .... ;

Success = DataSpaces->ReadTagged(  &MyGuid,  0,  RawData,
                                   sizeof(RawData),  NULL);
```

If you supply a *BufferSize* that is too small, **ReadTagged** will succeed but will write only the requested number of bytes to *Buffer*. If you specify a *BufferSize* that is too large, **ReadTagged** will succeed but will write only the actual block size to *Buffer*. If you supply a pointer for *TotalSize*, **ReadTagged** will use it to return the size of the actual block. If the block cannot be accessed, **ReadTagged** will return a failure status code.

If two blocks have identical GUID tags, the first matching block will be returned, and the second block will be inaccessible.

If you are not sure of the GUID tag of your block, you can use the **IDebugDataSpaces3::StartEnumTagged**, **IDebugDataSpaces3::GetNextTagged**, and **IDebugDataSpaces3::EndEnumTagged** methods to enumerate the tagged blocks. Their prototypes are as follows:

```
STDMETHOD(StartEnumTagged)(
    THIS_
    OUT PULONG64 Handle
    ) PURE;

STDMETHOD(GetNextTagged)(
    THIS_
    IN ULONG64 Handle,
    OUT LPGUID Tag,
    OUT PULONG Size
    ) PURE;

STDMETHOD(EndEnumTagged)(
    THIS_
    IN ULONG64 Handle
    ) PURE;
```

Debugging Callback Routines

It is also possible to debug the callback routine itself. Breakpoints within callback routines work just like any other breakpoint.

If the callback routine causes a second bug check, this new bug check will be processed first. However, Windows will not repeat certain parts of the Stop process—for example, it will not write a second crash dump file. The Stop code displayed on the blue screen will be the second bug check code. If a kernel debugger is attached, messages

about both bug checks will usually appear.

Debugging a User-Mode Failure with KD

8/27/2019 • 2 minutes to read • Edit Online

To properly debug user-mode failures, you need CDB or WinDbg. However, sometimes a user-mode exception will break into KD because no user-mode debugger is present. There are also times when it is helpful to monitor what specific user-mode processes are doing while debugging a kernel-mode problem.

By default, the kernel debugger attempts to load the first user-mode symbol that matches the address specified (for a **k**, **u**, or **ln** command).

Unfortunately, user-mode symbols are often not specified in the symbol path or the first symbol is not the correct one. If the symbols are not there, either copy them into the symbol path or use a **.sympath (Set Symbol Path)** command to point to the full symbol tree, and then use the **.reload (Reload Module)** command. If the wrong symbol is loaded, you can explicitly load a symbol by doing a **.reload <binary.ext>**.

Most of the Windows DLLs are rebased so they load at different addresses, but there are exceptions. Video adapters are the most common exceptions. There are dozens of video adapters that all load at the same base address, so KD will almost always find ati.dll (the first video symbol, alphabetically). For video, there is also a .sys file loaded that can be identified by using the **lm** command. With that information, you can issue a **.reload** to get the correct video DLLs. There are also times when the debugger gets confused and reloading specific symbols will help give the correct stack. Unassemble the functions to see if the symbols look correct.

Similar to the video DLLs, almost all executables load at the same address, so KD will report access. If you see a stack trace in access, do a **!process** and then a **.reload** of the executable name given. If the executable does not have symbols in the symbol path, copy them there and do the **.reload** again.

Sometimes KD or WinDbg has trouble loading the correct user-mode symbols even when the full symbol tree is in the symbol path. In this case, ntdll.dll and kernel32.dll are two of the most common symbols that would be required. In the case of debugging CSRSS from KD, winsrv.dll and csrsrv.dll are also common DLLs to load.

Crashing and Rebooting the Target Computer

12/5/2018 • 2 minutes to read • Edit Online

When you perform kernel debugging, you can cause the target computer to stop responding (that is, *crash* or *bug check*) by issuing the **.crash (Force System Crash)** command. This command immediately causes the target computer to stop responding. The debugger writes a kernel-mode dump file if you have enabled crash dumps. (For more information about these files, see Creating a Kernel-Mode Dump File.)

To restart the target computer, use the **.reboot (Reboot Target Computer)** command.

If you want the target computer to create a crash dump file and then restart, you should issue the **.crash** command, followed by the **.reboot** command. If you want only to restart, the **.crash** command is not required.

In the early stages of the boot process, the connection between the host computer and the target computer is lost. No information about the target computer is available to the debugger.

After the connection is broken, the debugger closes all symbol files and unloads all debugger extensions. At this point, all breakpoints are lost if you are running KD or CDB. In WinDbg, you can save the current workspace. This action saves all breakpoints.

If you want to end the debugging session at this point, use the **CTRL+B** command (in KD) or click **Exit** on the **File** menu (in WinDbg).

If you do not exit the debugger, the connection is reestablished after enough of the boot process has completed. Symbols and extensions are reloaded at this point. If you are running WinDbg, the kernel-mode workspace is reloaded.

You can tell the debugger to automatically break into the target computer during the restart process at two possible times:

- When the first kernel module is loaded into memory
- When the kernel initializes

To set an automatic breakpoint when the first kernel module loads, use the **-d** command-line option.

You can also change the break state after the debugger is running:

- Control the initial module load and kernel initialization breakpoints like all exceptions and events. You can break into the debugger when these events occur, or ignore them. You can also have a specified command automatically execute when these breakpoints are hit. For more information, see Controlling Exceptions and Events.
- Use the **CTRL+K** shortcut keys in KD, the CTRL+ALT+K shortcut keys in WinDbg, and the Debug | Kernel Connection | Cycle Initial Break command in WinDbg to change the break state. Every time that you use these commands, the debugger switches between three states: no automatic break, break upon kernel initialization, and break on first kernel module load. This method cannot activate both automatic breakpoints at the same time.

Mapping Driver Files

10/23/2019 • 6 minutes to read • Edit Online

Replacing driver files can be difficult. Frequently, you have to boot to the Microsoft Windows *safe build*, replace the driver binary, and then boot again.

However, Windows XP and later versions of Windows support a simpler method of replacing driver files. You can use this method to replace any kernel-mode driver (including display drivers), any Windows subsystem driver, or any other kernel-mode module. For simplicity, these files are called *drivers* in this topic, even though you can use this method for any kernel-mode module.

You can use this method whenever WinDbg or KD is attached as a kernel debugger. You can also use this method on a boot driver, but it is more difficult. For more information about how to use this method with boot drivers, see Replacing Boot Drivers.

To use a driver replacement map to replace driver files, do the following:

1. Create a *driver replacement map file*. This file is a text file that lists the drivers on the target computer and their replacement drivers on the host computer. You can replace any number of drivers. For example, you might create a file that is named Mymap.ini in the d:\Map_Files directory of your host computer that contains the following information.

   ```
   map
   \Systemroot\system32\drivers\videoprt.sys
   \\myserver\myshare\new_drivers\videoprt.sys
   ```

   For more information about the syntax of this file, see Driver Replacement Map File Format.

2. Set up a kernel debugging connection to the target computer, and start the kernel debugger (KD or WinDbg) on your host computer. (You do not have to actually break in to the target computer.)

3. Load the driver replacement map file by doing one of the following:

   - Set the _NT_KD_FILES environment variable before you start the kernel debugger.

     ```
     D:\Debugging Tools for Windows> set _NT_KD_FILES=d:\Map_Files\mymap.ini
     D:\Debugging Tools for Windows> kd
     ```

   - Use the **.kdfiles (Set Driver Replacement Map)** command after you start the kernel debugger.

     ```
     D:\Debugging Tools for Windows> kd
     kd> .kdfiles d:\Map_Files\mymap.ini
     KD file associations loaded from 'd:\Map_Files\mymap.ini'
     ```

     You can also use the **.kdfiles** command to display the current driver replacement map file or to delete the driver replacement map. If you do not use this command, the map persists until you exit the debugger.

After you complete this procedure, the driver replacement map takes effect.

Whenever the target computer is about to load a driver, it queries the kernel debugger to determine whether this driver has been mapped. If the driver has been mapped, the replacement file is sent over the kernel connection

and copied over the old driver file. The new driver is then loaded.

Driver Replacement Map File Format

Each driver file replacement is indicated by three lines in the driver replacement map file.

- The first line consists of the word "map".
- The second line specifies the path and file name of the old driver on the target computer.
- The third line specifies the full path of the new driver. This driver can be located on the host computer or on some other server.

You can repeat this pattern of information any number of times.

Paths and file names are case insensitive, and the actual driver file names can be different. The file that you specify on the third line is copied over the file that you specify on the second line when the target computer is about to load that driver.

Kdfiles will attempt to match the file name that is stored in the Service Control Manager (SCM) database. The name in the SCM database is identical to the name that was passed to MmLoadSystemImage.

In Windows 10 and later versions of the debugging tools, driver mapping works to match the driver name dynamically and determine the proper path. The full path does not need to be specified and the file extension is optional. You can use any of these entries to match the NT file system driver.

- ntfs
- NTFS
- ntfs.sys
- windows\system32\drivers\ntfs.sys

You can use any of these entries to match the NT kernel driver.

- ntoskrnl
- NTOSKRNL
- ntoskrnl.sys
- windows\system32\drivers\ntoskrnl.sys

The map file can include blank lines and can include comment lines that begin with a number sign (**#**). However, after "map" appears in the file, the next two lines must be the old driver and the new driver. The blank lines and comment lines cannot break up the three-line map blocks.

The following example shows a driver replacement map file.

```
map
\Systemroot\system32\drivers\videoprt.sys
e:\MyNewDriver\binaries\videoprt.sys
map
\Systemroot\system32\mydriver.sys
\\myserver\myshare\new_drivers\mydriver0031.sys

# Here is a comment
map
\??\c:\windows\system32\beep.sys
\\myserver\myshare\new_drivers\new_beep.sys
```

The driver replacement map file must be a text file, but you can use any file name and file name extension (.ini, .txt, .map, and so on).

Additional Notes

When driver substitution occurs, a message appears in the kernel debugger.

If you use **CTRL+D** (in KD) or CTRL+ALT+D (in WinDbg), you see verbose information about the replacement request. This information can be useful if you are not sure whether the name that you have listed matches the one in the SCM database.

You can enable the bcdedit bootdebug option to view early boot information that is useful for replacing the kernel, the hal, or boot drivers.

```
bcdedit -bootdebug on
```

For more information, see BCDEdit Options Reference.

If the kernel debugger exits, no more driver replacement occurs. However, any drivers that have already been replaced do not revert to their old binaries, because the driver files are actually overwritten.

This driver replacement feature automatically bypasses Windows File Protection (WFP).

You do not have to restart the target computer. Driver replacement occurs any time that the target computer loads a driver, regardless of whether it has been restarted. Of course, most drivers are loaded during the boot process, so in practice you should restart the target computer after the map file has been loaded.

If the _NT_KD_FILES variable is defined, the specified driver replacement map file is read when the kernel debugger is started. If you issue the **.kdfiles** command, the specified file is read immediately. At this point, the debugger verifies that the file has the basic map/line/line format. But the actual paths and file names are not verified until substitution occurs.

After the map file has been read, the debugger stores its contents. If you change this file after this point, the changes have no effect (unless you reissue the **.kdfiles** command).

Replacing Boot Drivers

If you want to replace a boot driver file by using this driver replacement method, you must connect the kernel debugger to the Windows boot loader (Ntldr), not to the Windows kernel. Before you can make this connection, you must install a special debugger-enabled version of Ntldr. You can find this version of Ntldr in the Windows Driver Kit (WDK), in the %DDKROOT%\debug directory.

Because the target computer bypasses its Boot.ini file, you cannot set the kernel connection protocol in the typical manner. You must make the connection through the COM1 port on the target computer. The baud rate is 115200. Therefore, the kernel debugger on the host computer should be configured to use a COM connection at the 115200 speed.

This special method applies only to boot drivers (that is, Acpi.sys, Classpnp.sys, Disk.sys, and anything else that **lm t n** displays at the initial Windows breakpoint). If you have to replace a standard driver that **MmLoadSystemImage** loads after the boot has been completed, you should use the standard method described earlier.

You cannot replace boot drivers on a computer that uses EFI firmware instead of the Boot.ini file.

Messages from the Target

12/5/2018 • 2 minutes to read • Edit Online

A target application or target computer can send messages to the debugger or break into the debugger, either conditionally or unconditionally, by using a variety of routines. A kernel-mode target can also test whether a debugger is currently attached.

This section includes:

- Breaking Into the Debugger
- Sending Output to the Debugger
- Reading and Filtering Debugging Messages
- Determining if a Debugger is Attached

Breaking Into the Debugger

6/25/2019 • 2 minutes to read • Edit Online

User-mode and kernel-mode code use different routines to break into the debugger.

User-Mode Break Routines

A break routine causes an exception to occur in the current process, so that the calling thread can signal the debugger associated with the calling process.

To break into a debugger from a user-mode program, use the DebugBreak function.

When a user-mode program calls **DebugBreak**, the following possible actions will occur:

1. If a user-mode debugger is attached, the program will break into the debugger. This means that the program will pause and the debugger will become active.
2. If no user-mode debugger is attached, but kernel-mode debugging was enabled at boot time, the entire computer will break into the kernel debugger. If no kernel debugger is attached, the computer will freeze and await a kernel debugger.
3. If no user-mode debugger is attached, and kernel-mode debugging is not enabled, the program will terminate with an unhandled exception, and the post-mortem (just-in-time) debugger will be activated. For more information, see Enabling Postmortem Debugging.

Kernel-Mode Break Routines

When a kernel-mode program breaks into the debugger, the entire operating system freezes until the kernel debugger allows execution to resume. If no kernel debugger is present, this is treated as a bug check.

The **DbgBreakPoint** routine works in kernel-mode code, but is otherwise similar to the **DebugBreak** user-mode routine.

DbgBreakPointWithStatus also causes a break, but it additionally sends a 32-bit status code to the debugger.

KdBreakPoint and **KdBreakPointWithStatus** are identical to **DbgBreakPoint** and **DbgBreakPointWithStatus**, respectively, when compiled in the checked build environment. When compiled in the free build environment, they have no effect.

For complete documentation of these routines, as well as the build environment, see the Windows Driver Kit.

Kernel-Mode Conditional Break Routines

Two conditional break routines are available for kernel-mode code. These routines test a logical expression. If the expression is false, execution halts and the debugger becomes active.

The **ASSERT** macro causes the debugger to display the failed expression and its location in the program. The **ASSERTMSG** macro is similar, but allows an additional message to be sent to the debugger.

ASSERT and **ASSERTMSG** are only active when compiled in the checked build environment. When compiled in the free build environment, they have no effect.

For complete documentation of these routines, as well as the build environment, see the Windows Driver Kit.

Sending Output to the Debugger

12/5/2018 • 2 minutes to read • Edit Online

User-mode and kernel-mode code use different routines to send output to the debugger.

User-Mode Output Routines

OutputDebugString sends a null-terminated string to the debugger of the calling process. In a user-mode driver, **OutputDebugString** displays the string in the Debugger Command window. If a debugger is not running, this routine has no effect. **OutputDebugString** does not support the variable arguments of a **printf** formatted string.

For complete documentation of this routine, see the Microsoft Windows SDK.

Kernel-Mode Output Routines

DbgPrint displays output in the debugger window. This routine supports the basic **printf** format parameters. Only kernel-mode code can call **DbgPrint**.

DbgPrintEx is similar to **DbgPrint**, but it allows you to "tag" your messages. When running the debugger, you can permit only those messages with certain tags to be sent. This allows you to view only those messages that you are interested in. For details, see Reading and Filtering Debugging Messages.

Note In Windows Vista and later versions of Windows, **DbgPrint** produces tagged messages as well. This is a change from previous versions of Windows.

KdPrint and **KdPrintEx** are identical to **DbgPrint** and **DbgPrintEx**, respectively, when compiled in the checked build environment. When compiled in the free build environment, they have no effect.

For complete documentation of these routines, as well as the build environment, see the Windows Driver Kit.

Reading and Filtering Debugging Messages

12/5/2018 • 10 minutes to read • Edit Online

Kernel-mode code can use the **DbgPrintEx** and **KdPrintEx** routines to send a messages to the kernel debugger that are only transmitted under certain conditions. This allows you to filter out messages that you are not interested in.

Note In Windows Server 2003 and earlier versions of Windows, **DbgPrint** and **KdPrint** send messages to the kernel debugger unconditionally. In Windows Vista and later versions of Windows, these routines send messages conditionally, like **DbgPrintEx** and **KdPrintEx**. Whichever version of Windows you are using, it is recommended that you use **DbgPrintEx** and **KdPrintEx**, since these allow you to control the conditions under which the message will be sent.

For complete documentation of these routines, see the Windows Driver Kit.

The basic procedure is as follows:

To filter debugging messages

1. For each message you wish to send to the debugger, use the function **DbgPrintEx** or **KdPrintEx** in your driver's code. Pass the appropriate component name to the *ComponentId* parameter, and pass a value to the *Level* parameter that reflects the severity or nature of this message. The message itself is passed to the *Format* and *arguments* parameters as with **printf**.

2. Set the value of the appropriate *component filter mask*. Each component has a different mask; the mask value indicates which of that component's messages will be displayed. The component filter mask may be set in the registry using a registry editor, or in memory using a kernel debugger.

3. Attach a kernel debugger to the computer. Each time your driver passes a message to **DbgPrintEx** or **KdPrintEx**, the values passed to *ComponentId* and *Level* will be compared with the value of the corresponding component filter mask. If these values satisfy certain criteria, the message will be sent to the kernel debugger and displayed. Otherwise, no message will be sent.

Full details follow. All references on this page to **DbgPrintEx** apply equally to **KdPrintEx**.

Identifying the Component Name

Each component has a separate filter mask. This allows the debugger to configure the filter for each component separately.

Each component is referred to in different ways, depending on the context. In the *ComponentId* parameter of **DbgPrintEx**, the component name is prefixed with "DPFLTR_" and suffixed with "_ID". In the registry, the component filter mask has the same name as the component itself. In the debugger, the component filter mask is prefixed with "Kd_" and suffixed with "_Mask".

There is a complete list of all component names (in DPFLTR_*XXXX*_ID format) in the Microsoft Windows Driver Kit (WDK) header ntddk.h and the Windows SDK header ntrtl.h. Most of these component names are reserved for Windows and for drivers written by Microsoft.

There are six component names reserved for independent hardware vendors. To avoid mixing your driver's output with the output of Windows components, you should use one of the following component names:

COMPONENT NAME	DRIVER TYPE
IHVVIDEO	Video driver
IHVAUDIO	Audio driver
IHVNETWORK	Network driver
IHVSTREAMING	Kernel streaming driver
IHVBUS	Bus driver
IHVDRIVER	Any other type of driver

For example, if you are writing a video driver, you would use DPFLTR_IHVVIDEO_ID as the *ComponentId* parameter of **DbgPrintEx**, use the value name **IHVVIDEO** in the registry, and refer to **Kd_IHVVIDEO_Mask** in the debugger.

In Windows Vista and later versions of Windows, all messages sent by **DbgPrint** and **KdPrint** are associated with the **DEFAULT** component.

Choosing the Correct Level

The *Level* parameter of the **DbgPrintEx** routine is of type DWORD. It is used to determine the *importance bit field*. The connection between the *Level* parameter and this bit field depends on the size of *Level*:

- If *Level* is equal to a number between 0 and 31, inclusive, it is interpreted as a bit shift. The importance bit field is set to the value 1 << *Level*. Thus choosing a value between 0 and 31 for *Level* results in a bit field with exactly one bit set. If *Level* is 0, the bit field is equivalent to 0x00000001;if *Level* is 31, the bit field is equivalent to 0x80000000.
- If *Level* is a number between 32 and 0xFFFFFFFF inclusive, the importance bit field is set to the value of *Level* itself.

Thus, if you wish to set the bit field to 0x00004000, you can specify *Level* as 0x00004000 or simply as 14. Note that certain bit field values are not possible by this system -- including a bit field which is entirely zero.

The following constants can be useful for setting the value of *Level*. They are defined in the Microsoft Windows Driver Kit (WDK) header ntddk.h and the Windows SDK header ntrtl.h:

```
#define   DPFLTR_ERROR_LEVEL     0
#define   DPFLTR_WARNING_LEVEL   1
#define   DPFLTR_TRACE_LEVEL     2
#define   DPFLTR_INFO_LEVEL      3
#define   DPFLTR_MASK    0x8000000
```

One easy way to use the *Level* parameter is to always use values between 0 and 31 -- using the bits 0, 1, 2, 3 with the meaning given by DPFLTR_*XXXX*_LEVEL, and using the other bits to mean whatever you choose.

Another easy way to use the *Level* parameter is to always use explicit bit fields. If you choose this method, you may wish to OR the value DPFLTR_MASK with your bit field; this assures that you will not accidentally use a value less than 32.

To make your driver compatible with the way Windows uses message levels, you should only set the lowest bit (0x1) of the importance bit field if a serious error occurs. If you are using *Level* values less than 32, this corresponds to DPFLTR_ERROR_LEVEL. If this bit is set, your message is going to be viewed any time someone attaches a kernel debugger to a computer on which your driver is running.

The warning, trace, and information levels should be used in the appropriate situations. Other bits can be freely used for any purposes that you find useful. This allows you to have a wide variety of message types that can be selectively seen or hidden.

In Windows Vista and later versions of Windows, all messages sent by **DbgPrint** and **KdPrint** behave like **DbgPrintEx** and **KdPrintEx** messages with *Level* equal to DPFLTR_INFO_LEVEL. In other words, these messages have the third bit of their importance bit field set.

Setting the Component Filter Mask

There are two ways to set a component filter mask:

- The component filter mask can be accessed in the registry key **HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Debug Print Filter**. Using a registry editor, create or open this key. Under this key, create a value with the name of the desired component, in uppercase. Set it equal to the DWORD value that you wish to use as the component filter mask.

- If a kernel debugger is active, it can access the component filter mask value by dereferencing the address stored in the symbol **Kd**_*XXXX*_**Mask**, where *XXXX* is the desired component name. You can display the value of this mask in WinDbg or KD with the **dd (Display DWORD)** command, or enter a new component filter mask with the **ed (Enter DWORD)** command. If there is a danger of symbol ambiguity, you may wish to specify this symbol as **nt!Kd**_*XXXX*_**Mask**.

Filter masks stored in the registry take effect during boot. Filter masks created by the debugger take effect immediately, and persist until Windows is rebooted. A value set in the registry can be overridden by the debugger, but the component filter mask will return to the value specified in the registry if the system is rebooted.

There is also a system-wide mask called **WIN2000**. This is equal to 0x1 by default, though it can be changed through the registry or the debugger like all other components. When filtering is performed, each component filter mask is first ORed with the **WIN2000** mask. In particular, this means that components whose masks have never been specified default to 0x1.

Criteria for Displaying the Message

When **DbgPrintEx** is called in kernel-mode code, Windows compares the message importance bit field specified by *Level* with the filter mask of the component specified by *ComponentId*.

Note Recall that when the *Level* parameter is between 0 and 31, the importance bit field is equal to 1 << *Level*, but when the *Level* parameter is 32 or higher, the importance bit field is simply equal to *Level*.

Windows performs an AND operation on the importance bit field and the component filter mask. If the result is nonzero, the message is sent to the debugger.

Example

Here is an example.

Suppose that before the last boot, you created the following values in the **Debug Print Filter** key:

- **IHVVIDEO**, with a value equal to DWORD 0x2

- **IHVBUS**, equal to DWORD 0x7FF

Now you issue the following commands in the kernel debugger:

```
kd> ed Kd_IHVVIDEO_Mask 0x8
kd> ed Kd_IHVAUDIO_Mask 0x7
```

At this point, the **IHVVIDEO** component has a filter mask of 0x8, the **IHVAUDIO** component has a filter mask of 0x7, and the **IHVBUS** component has a filter mask of 0x7FF.

However, because these masks are automatically ORed with the **WIN2000** system-wide mask (which is usually equal to 0x1), the **IHVVIDEO** mask is effectively equal to 0x9. Indeed, components whose filter masks have not been set at all (for instance, **IHVSTREAMING** or **DEFAULT**) will have a filter mask of 0x1.

Now suppose that the following function calls occur in various drivers:

```
DbgPrintEx( DPFLTR_IHVVIDEO_ID,  DPFLTR_INFO_LEVEL,   "First message.\n");
DbgPrintEx( DPFLTR_IHVAUDIO_ID,  7,                   "Second message.\n");
DbgPrintEx( DPFLTR_IHVBUS_ID,    DPFLTR_MASK | 0x10,  "Third message.\n");
DbgPrint( "Fourth message.\n");
```

The first message has its *Level* parameter equal to DPFLTR_INFO_LEVEL, which is 3. Since this is less than 32, it is treated as a bit shift, resulting in an importance bit field of 0x8. This value is then ANDed with the effective **IHVVIDEO** component filter mask of 0x9, giving a nonzero result. So the first message is transmitted to the debugger.

The second message has its *Level* parameter equal to 7. Again, this is treated as a bit shift, resulting in an importance bit field of 0x80. This is then ANDed with the **IHVAUDIO** component filter mask of 0x7, giving a result of zero. So the second message is not transmitted.

The third message has its *Level* parameter equal to DPFLTR_MASK | 0x10. This is greater than 31, and therefore the importance bit field is set equal to the value of *Level* -- in other words, to 0x80000010. This is then ANDed with the **IHVBUS** component filter mask of 0x7FF, giving a nonzero result. So the third message is transmitted to the debugger.

The fourth message was passed to **DbgPrint** instead of **DbgPrintEx**. In Windows Server 2003 and earlier versions of Windows, messages passed to this routine are always transmitted. In Windows Vista and later versions of Windows, messages passed to this routine are always given a default filter. The importance bit field is equal to 1 << DPFLTR_INFO_LEVEL, which is 0x00000008. The component for this routine is **DEFAULT**. Since you have not set the **DEFAULT** component filter mask, it has a value of 0x1. When this is ANDed with the importance bit field, the result is zero. So the fourth message is not transmitted.

The DbgPrint Buffer

When **DbgPrint**, **DbgPrintEx**, **KdPrint**, or **KdPrintEx** transmits a message to the debugger, the formatted string is sent to the *DbgPrint buffer*. In most cases, the contents of this buffer are displayed immediately in the Debugger Command window. This display can be disabled by using the **Buffer DbgPrint Output** option of the Global Flags Utility (gflags.exe). This display does not automatically appear during local kernel debugging.

During local kernel debugging, and any other time this display has been disabled, the contents of the DbgPrint buffer can only be viewed by using the **!dbgprint** extension command.

Any single call to **DbgPrint**, **DbgPrintEx**, **KdPrint**, or **KdPrintEx** will only transmit 512 bytes of information. Any output longer than this will be lost. The DbgPrint buffer itself can hold up to 4 KB of data on a free build of Windows, and up to 32 KB of data on a checked build of Windows. On Windows Server 2003 and later versions of Windows, you can use the KDbgCtrl tool to alter the size of the DbgPrint buffer. See Using KDbgCtrl for details.

If a message is filtered out because of its *ComponentId* and *Level* values, it is not transmitted across the debugging connection. Therefore there is no way to display this message in the debugger.

Determining if a Debugger is Attached

1/3/2019 • 2 minutes to read • Edit Online

Kernel-mode code can determine the status of kernel debugging by using the following variables and routines:

- The KD_DEBUGGER_ENABLED global kernel variable indicates whether kernel debugging is enabled.
- The KD_DEBUGGER_NOT_PRESENT global kernel variable indicates whether a kernel debugger is currently attached.
- (Windows Server 2003 and later) The **KdRefreshDebuggerNotPresent** routine refreshes the value of KD_DEBUGGER_NOT_PRESENT.

For complete documentation, see the Windows Driver Kit.

Specialized Debugging Techniques

12/5/2018 • 2 minutes to read • Edit Online

This section describes debugging techniques that apply to particular technologies and types of code modules.

You can learn more in the following topics.

- Windows Runtime Debugging
- Kernel-Mode Driver Framework Debugging
- User-Mode Driver Framework Debugging
- Debugging Managed Code
- Debugging Device Nodes and Device Stacks
- Debugging Plug and Play and Power Issues
- Debugging a User-Mode Failure with KD
- Debugging a Device Installation Co-Installer
- Debugging a Dual-Boot Machine
- Debugging Windows Setup and the OS Loader
- Debugging CSRSS
- Debugging WinLogon
- Debugging BIOS Code
- Specifying Module and Function Owners

Debugging on ARM64

6/25/2019 • 2 minutes to read • Edit Online

This topic describes debugging Windows 10 on ARM Processors. For general information about Windows 10 on ARM, see Windows 10 desktop on ARM64.

In general, developers debugging user mode apps should use the version of the debugger that matches the architecture of the target app. Use the ARM64 version of WinDbg to debug user mode ARM64 applications and use the ARM version of WinDbg to debug user mode ARM32 applications. Use the x86 version of WinDbg to debug user mode x86 applications running on ARM64 processors.

In rare cases where you need to debug system code – such as WOW64 or CHPE – you can use the ARM64 version of WinDbg. If you are debugging the ARM64 kernel from another machine, use the version of WinDbg that matches the architecture of that other machine.

Getting ARM Debugging Tools for Windows

You can get debugging tools for ARM64 by downloading the Windows 10 SDK (version 10.0.16299 or later). During the installation, select the *Debugging Tools for Windows* box.

The debugging tools are located in the `Debuggers` folder in the kit installation directory. The x86 tools are under `Debuggers\x86`, the ARM32 tools are under `Debuggers\ARM`, and the ARM64 tools are under `Debuggers\ARM64`.

Debugging ARM64 Code

ARM64 WinDbg is required to debug ARM64 code. The debugging experience is similar to debugging x86 applications with x86 WinDbg on x86 Windows, except for the following differences.

- There are 32 general purpose registers - x0 to x28 and fp, lr, sp.
- Program counter register, pc, is not a general purpose register.
- All general purpose registers and pc register are 64-bit in width.
- At most 2 active data breakpoints for execution and 2 active data breakpoints for read/write memory. For more information, see Processor Breakpoints.

Debugging x86 User Mode Code

In the rare cases that you need to use ARM64 WinDbg to debug your x86 user mode code, you can use the following WinDbg commands to switch between contexts:

- .effmach x86: Switch to and see x86 context, simulating the effect of using x86 WinDbg.
- .effmach arm64: Switch to and see ARM64 context
- .effmach chpe: Switch to and see CHPE context.

For more information about the .effmach, see .effmach (Effective Machine).

When debugging x86 apps in user mode, regardless of which WinDbg version you are using, be aware of these considerations.

- If a thread is not being actively debugged (e.g. single-stepped, encountered a breakpoint), not reporting an exception, and not in a system call, the register context may not be up-to-date.
- The emulator internally generates Data misaligned, Illegal instruction, In-page I/O error exceptions and handles the ones it generates. When you are using WinDbg, consider configuring these exceptions as *Ignored* under the

Debug / Event Filters... menu item.

- If using ARM64 WinDbg in user mode, single-stepping across x86 & CHPE function boundaries is not supported. To work around this, set breakpoints on the target code.

For general information about ARM64 and WOW64 see Running 32-bit Applications in the 64-bit Windows programming guide.

For information on debugging applications running under WOW64, see Debugging WOW64.

Debugging in Visual Studio

For information on debugging ARM in Visual Studio, see Remote Debugging.

See Also

Building ARM64 Drivers with the WDK

Windows Community Standup discussing the Always Connected PC

Windows Runtime Debugging

12/5/2018 • 2 minutes to read • Edit Online

You can use the following debugger extensions to debug code that uses data types defined by the Windows Runtime.

- **!hstring**
- **!hstring2**
- **!winrterr**

Kernel-Mode Driver Framework Debugging

12/5/2018 • 2 minutes to read • Edit Online

Debugging extensions for Kernel-Mode Driver Framework (KMDF) are contained in the Wdfkd.dll extension library.

You can use the extension commands that the Wdfkd.dll extension library contains to debug drivers that use KMDF.

For a description of the extension commands in Wdfkd.dll, see Kernel-Mode Driver Framework Extensions (Wdfkd.dll).

These extensions can be used on Microsoft Windows XP and later operating systems. Some extensions have additional restrictions; these restrictions are noted on the individual reference pages.

Note When you create a new KMDF or UMDF driver, you must select a driver name that has 32 characters or less. This length limit is defined in wdfglobals.h. If your driver name exceeds the maximum length, your driver will fail to load.

To use this extension library, you must load the library into your debugger. For information about how to load extension libraries into a debugger, see Loading Debugger Extension DLLs.

User-Mode Driver Framework Debugging

1/3/2019 • 2 minutes to read • Edit Online

For an overview of how to debug User-Mode Driver Framework (UMDF) drivers, including information on how to start this kind of debugging session, see the Debugging UMDF Drivers section of the Windows Driver Kit (WDK) documentation.

UMDF Debugging Extensions

User-Mode Driver Framework (UMDF) debugging extensions are implemented in the extension module Wudfext.dll. You can use these extensions to debug drivers that use UMDF.

For a complete description of the extension commands in Wudfext.dll, see User-Mode Driver Framework Extensions (Wudfext.dll).

Some extensions have additional restrictions on the Windows version or UMDF version that is required; these restrictions are noted on the individual reference pages.

Note When you create a new KMDF or UMDF driver, you must select a driver name that has 32 characters or less. This length limit is defined in wdfglobals.h. If your driver name exceeds the maximum length, your driver will fail to load.

To use this extension library, you must load the library into your debugger. For information about how to load extension libraries into a debugger, see Loading Debugger Extension DLLs.

Debugging Device Nodes and Device Stacks

12/5/2018 • 2 minutes to read • Edit Online

You can use the following commands for debugging device nodes and device stacks.

- **!devnode**
- **!devstack**
- **!devobj**
- **!drvobj**

Debugging Plug and Play and Power Issues

12/5/2018 • 2 minutes to read • Edit Online

The following extensions are useful for debugging Plug and Play and power issues.

- **!pnpevent**
- **!pocaps**
- **!popolicy**

Debugging a User-Mode Failure with KD

8/27/2019 • 2 minutes to read • Edit Online

To properly debug user-mode failures, you need CDB or WinDbg. However, sometimes a user-mode exception will break into KD because no user-mode debugger is present. There are also times when it is helpful to monitor what specific user-mode processes are doing while debugging a kernel-mode problem.

By default, the kernel debugger attempts to load the first user-mode symbol that matches the address specified (for a **k**, **u**, or **ln** command).

Unfortunately, user-mode symbols are often not specified in the symbol path or the first symbol is not the correct one. If the symbols are not there, either copy them into the symbol path or use a **.sympath (Set Symbol Path)** command to point to the full symbol tree, and then use the **.reload (Reload Module)** command. If the wrong symbol is loaded, you can explicitly load a symbol by doing a **.reload <binary.ext>**.

Most of the Windows DLLs are rebased so they load at different addresses, but there are exceptions. Video adapters are the most common exceptions. There are dozens of video adapters that all load at the same base address, so KD will almost always find ati.dll (the first video symbol, alphabetically). For video, there is also a .sys file loaded that can be identified by using the **lm** command. With that information, you can issue a **.reload** to get the correct video DLLs. There are also times when the debugger gets confused and reloading specific symbols will help give the correct stack. Unassemble the functions to see if the symbols look correct.

Similar to the video DLLs, almost all executables load at the same address, so KD will report access. If you see a stack trace in access, do a **!process** and then a **.reload** of the executable name given. If the executable does not have symbols in the symbol path, copy them there and do the **.reload** again.

Sometimes KD or WinDbg has trouble loading the correct user-mode symbols even when the full symbol tree is in the symbol path. In this case, ntdll.dll and kernel32.dll are two of the most common symbols that would be required. In the case of debugging CSRSS from KD, winsrv.dll and csrsrv.dll are also common DLLs to load.

Debugging a Device Installation Co-Installer

12/5/2018 • 3 minutes to read • Edit Online

Some hardware device installation packages include DLL files known as *co-installers*, which assist with installing the device.

You cannot debug a co-installer in the same fashion as other modules. This is because of the unique way in which a co-installer is loaded, and because many installation scenarios occur automatically without providing the developer an opportunity to break into the running process.

You can resolve this issue by programmatically installing the device. Attaching a debugger to the application which installs the device allows access to the co-installer itself. The simplest way to accomplish this is to install or reinstall the device using the DevCon tool that is included in the Windows Driver Kit (WDK). You can then debug the co-installer with WinDbg.

Use the following procedure to accomplish this task. This procedure assumes you have developed a working driver installation package for your device which uses a co-installer. It also assumes that you have the latest copy of the WDK. For information on developing drivers, driver installation packages, and driver installation co-installers, see the WDK documentation.

Debugging a Co-installer Using DevCon and WinDbg

1. Plug in the hardware device.
2. Cancel the **New Hardware Found** wizard.
3. Start WinDbg.
4. Select **Open Executable** from WinDbg's **File** menu.
5. In the **Open Executable** dialog box, do the following:
   a. In the file selection text box, select the DevCon tool (Devcon.exe). For this, browse to the WDK installation folder, then open the subdirectory tools, then open the subdirectory devcon, then open the subdirectory that matches the processor architecture of your machine, and then select Devcon.exe. Click only once on Devcon.exe and do not yet press **Open**.
   b. In the **Arguments** text box, enter the following text, where *INFFile* is the filename of your Device Installation Information (INF) file, and *HardwareID* is the hardware ID of your device:

      ```
      update INFFile HardwareID
      ```

   c. In the **Start directory** text box, enter the path to your device installation package.
   d. Click **Open**.
6. The debugging process will begin, and WinDbg will break into the DevCon process before DevCon installs your driver.
7. Configure the debugger to break into the co-installer process when it is loaded. You can do this by either of the following methods:
   - In the Debugger Command window, use the **sxe (Set Exceptions)** command followed by **Id:** and then the filename of the co-installer, excluding the file extension. There should be no space after the colon For example, if the name of the co-installer is mycoinst.dll, you would use the following

command:

```
sxe ld:mycoinst
```

- Select **Event Filters** from WinDbg's **Debug** menu. In the **Event Filters** dialog box, select **Load module**. Under **Execution**, select **Enabled.** Under **Continue**, select **Not Handled.** Click the **Argument** button, and then in the text box enter the filename of the co-installer, excluding the file extension (for example, enter "mycoinst" for mycoinst.dll). Click **OK** and then click **Close**.

8. Resume execution by pressing F5 or entering the **g (Go)** command in the Debugger Command window.
9. When the co-installer is loaded, execution will break back into the debugger. At this point, you can set any additional breakpoints that you need.

Limitations of This Procedure

In certain cases, running a device installation package under DevCon may result in slightly different behavior than that of a PnP installation, because of different security tokens and the like. If you are trying to debug a specific problem in your co-installer, it is possible that this problem will not replicate if DevCon is involved. Therefore, before using this technique, you should use DevCon to install your driver without a debugger attached to verify that this problem exists in both the PnP and the DevCon scenarios.

If the problem vanishes whenever DevCon initiates the installation, then you will have to debug your co-installer without using DevCon. One way of doing this is to use the TList tool with the **/m** option to determine which process is loading the co-installer module, and then attaching the debugger to that process.

Debugging a Dual-Boot Machine

12/5/2018 • 2 minutes to read • Edit Online

How should you respond when the alternate operating system does not start on a dual-boot machine?

First, check that the boot options point to the correct path for the other operating system. See Getting Set Up for Debugging for details.

On an x86 computer, you should also verify that boosect.ini exists. This file contains the boot record for the other operating system. To unhide this file, use the **attrib -r -s -h boosect.ini** command.

Debugging Windows Setup and the OS Loader

10/23/2019 • 2 minutes to read • Edit Online

Debugging During the Boot Sequence

For information about debugging a computer as it boots, see the following topics.

- **BCDEdit /bootdebug**
- BCD Boot Options Reference

Debugging CSRSS

9/16/2019 • 2 minutes to read • Edit Online

The Client Server Run-Time Subsystem (CSRSS) is the user-mode process that controls the underlying layer for the Windows environment. There are a number of problems that make it necessary to debug CSRSS itself.

Debugging CSRSS is also useful when the Windows subsystem terminates unexpectedly with a **Bug Check 0xC000021A** (WINLOGON_FATAL_ERROR). In this case, debugging CSRSS will catch the failure before it gets to an "unexpected" point.

Controlling NTSD from the Kernel Debugger

The easiest way to debug CSRSS is to use NTSD and control it from the kernel debugger.

Enabling CSRSS Debugging

CSRSS debugging must be enabled before you can proceed. If the target computer is running a *checked build* of Windows, CSRSS debugging is always enabled. If the target computer is running a *free build* of Windows, you will have to enable CSRSS debugging through the Global Flags Utility (GFlags).

To do this, start the GFlags utility, select the **System Registry** radio button, and select **Enable debugging of Win32 subsystem**.

Alternatively, you can use the following GFlags command-line:

```
gflags /r +20000
```

Or, if you prefer, you can edit the registry key manually instead of using GFlags. Open the following registry key:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager
```

Edit the **GlobalFlag** value entry (of type REG_DWORD) and set the bit 0x00020000.

After using GFlags or manually editing the registry, you must reboot for the changes to take effect.

Starting NTSD

Because you will be controlling the user-mode debugger from the kernel debugger, you will need to set up a kernel debugging connection. See Getting Set Up for Debugging for details.

After the registry has been properly configured, it is a simple matter of starting NTSD as follows:

ntsd --

See Controlling the User-Mode Debugger from the Kernel Debugger for an explanation of how to proceed.

You will have to set your symbol path to a location on your host computer or to some other location on your network. When CSRSS is being debugged, network authentication on the target computer will not work properly.

Note that you may see an "in page io error" message. This is another manifestation of a hardware failure.

When the debugging session ends, the debugger will detach from CSRSS while the CSRSS process is still running. This avoids termination of the CSRSS process itself.

Debugging WinLogon

12/5/2018 • 2 minutes to read • Edit Online

WinLogon is the user-mode process that handles the task of interactive users logging on and logging off, and handles all instances of CTRL+ALT+DELETE.

Controlling NTSD from the Kernel Debugger

The easiest way to debug WinLogon is to use NTSD and control it from the kernel debugger.

Enabling WinLogon Debugging

Because you will be redirecting the user-mode debugger output to the kernel debugger, you will need to set up a kernel debugging connection. See Getting Set Up for Debugging.

To attach a debugger to WinLogon, you must go through the registry so the process is debugged from the time it starts up. To set up WinLogon debugging, set **HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\WinLogon.EXE\Debugger** to:

```
ntsd -d -x -g
```

The **-d** option passes control to the kernel debugger. The **-x** option causes the debugger to capture access violations as second-chance exceptions. The **-g** option causes the WinLogon process to run after the attachment. Do not add the **-g** if you want to start debugging before Winlogon.exe begins (for example, if you want to set an initial breakpoint).

In addition, you should set the GlobalFlag value under the **winlogon.exe** key to REG_DWORD "0x000400F0". This sets heap checking and FLG_ENABLE_KDEBUG_SYMBOL_LOAD. However, since this second flag only affects the kernel debugger, symbols must also be copied to the target computer before starting the debugger.

The registry change requires a reboot to take effect.

Performing the Debugging

After the next reboot, the debugger will break into WinLogon automatically.

See Controlling the User-Mode Debugger from the Kernel Debugger for an explanation of how to proceed.

You will have to set your symbol path to a location on your host computer or to some other location on your network. When WinLogon is being debugged, network authentication on the target computer will not work properly.

Debugging BIOS Code

12/5/2018 • 2 minutes to read • Edit Online

BIOS code is not built from standard assembly code, so it requires different debugging techniques.

On an x86-based processor, the BIOS uses 16-bit code. To disassemble this code, use the **ux (Unassemble x86 BIOS)** command. To display information about the Intel Multiprocessor Specification (MPS), use the **!mps** extension.

If you are debugging ACPI BIOS code, the preceding commands do not work, because ACPI BIOS is written in ACPI Machine Language (AML). To disassemble this code, you should use **!amli u**. For more information about this kind of debugging, see ACPI Debugging.

Specifying Module and Function Owners

7/9/2019 • 4 minutes to read • Edit Online

The **!analyze** and **!owner** extensions use a file that is named Triage.ini to determine the owner of the symbols that the debugger encounters.

When you use these extensions, the identities of the function or module owner are displayed after the word "Followup".

The Triage.ini file is a text file that resides in the \triage subdirectory of your Debugging Tools for Windows installation. A sample Triage.ini file is included as part of the Debugging Tools for Windows package.

Warning If you install an updated version of Debugging Tools for Windows in the same directory as the current version, it overwrites all of the files in that directory, including Triage.ini. After you modify or replace the sample Triage.ini file, save a copy of it to a different directory. After you reinstall the debuggers, you can copy the saved Triage.ini over the default version.

Format of the Triage.ini File

Although the Triage.ini file is intended to help you determine the owner of a function that has broken into the debugger, the "owner" strings in this file can be anything that might help you with debugging. The strings can be names of people who wrote or maintain the code. Or, the strings can be short instructions about what you can do when an error occurs in a module or function.

Each line in this file has the following syntax.

```
Module[!Function]=Owner
```

You can add an asterisk (*) only at the end of a module or function name. If it appears elsewhere, it is interpreted as a literal character.

You cannot add spaces in the owner string. If spaces do exist in the owner string, they are ignored.

For more information about syntax options, see Special Triage.ini Syntax.

The following examples shows a sample Triage.ini file.

```
module1=Person1
module2!functionA=Person2
module2!functionB=Person3
module2!funct*=Person4
module2!*=Person5
module3!singleFunction=Person6
mod*!functionC=Person7
```

Triage.ini and !owner

When you pass a module or function name to the **!owner** extension, the debugger displays the word "Followup" followed by the name of the module or function's owner.

The following example uses the previous sample Triage.ini file.

```
0:000> !owner module2!functionB
Followup:  Person3
```

According to the file, "Person3" owns **module2!functionB**, and "Person4" owns **module2!funct***. *Both of these strings match the argument that is passed to* ***!owner****, so the more complete match is used.*

Triage.ini and !analyze

When you use the ***!analyze*** *extension, the debugger looks at the top faulting frame in the stack and tries to determine the owner of the module and function in this frame. If the debugger can determine the owner, the owner information is displayed.*

If the debugger cannot determine the owner, the debugger passes to the next stack frame, and so on, until the debugger determines the owner or the stack is completely examined.

If the debugger can determine the owner, the owner name is displayed after the word "Followup". If the debugger searches the whole stack without finding any information, no name is displayed.

The following example uses the sample Triage.ini file that is given earlier in this topic.

Suppose the first frame on the stack is ***MyModule!someFunction****. The debugger does not find* ***MyModule*** *in the Triage.ini file. Next, it continues to the second frame on the stack.*

Suppose the second frame is ***module3!anotherFunction****. The debugger does see an entry for* ***module3****, but there is no match for* ***anotherFunction*** *in this module. Next, the debugger continues to the third frame.*

Suppose the third frame is ***module2!functionC****. The debugger first looks for an exact match, but such a match does not exist. The debugger then trims the function name and discovers* ***module2!funct**** *in Triage.ini. This match ends the search, because the debugger determines that the owner is "Person4".*

The debugger then displays output that is similar to the following example.

```
0:000> !analyze
*******************************************************************************
*                                                                             *
*                        Exception Analysis                                   *
*                                                                             *
*******************************************************************************

Use !analyze -v to get detailed debugging information.

Probably caused by : module2 ( module2!functionC+15a )

Followup: Person4
---------
```

A more complete match takes precedence over a shorter match. However, a module name match is always preferred to a function name match. If ***module2!funct**** *had not been in this Triage.ini file, the debugger would have selected* ***module2!**** *as the match. And if both* ***module2!funct**** *and* ***module2!**** *were removed,* ***mod*!functionC*** *would have been selected.*

Special Triage.ini Syntax

If you omit the exclamation point and function name or add ***!**** *after a module name, all functions in that module are indicated. If a function within this module is also specified separately, the more precise specification takes precedence.*

If you use "default" as a module name or a function name, it is equivalent to a wildcard character. For example, ***nt!**** *is the same as* ***nt!default****, and* ***default*** *is the same as* ****!****.

If a match is made, but the word ***ignore*** *appears to the right of the equal sign (=), the debugger continues to the next frame in the stack.*

You can add ***last_*** *or* ***maybe_*** *before an owner's name. This prefix gives the owner less priority when you run*

!analyze. *The debugger chooses a definite match that is lower on the stack over a* ***maybe_*** *match that is higher on the stack. The debugger also chooses a* ***maybe_*** *match that is lower on the stack over a* ***last_*** *match that is higher on the stack.*

Sample Triage.ini

A sample Triage.ini template is included in the Debugging Tools for Windows package. You can add the owners of any modules and functions that you want to this file. If you want to have no global default, delete the ***default=MachineOwner*** *line at the beginning of this file.*

RPC Debugging

12/5/2018 • 2 minutes to read • Edit Online

This section includes:

Overview of RPC Debugging

Enabling RPC State Information

Displaying RPC State Information

Common RPC Debugging Techniques

RPC State Information Internals

Overview of RPC Debugging

7/13/2019 • 2 minutes to read • Edit Online

Microsoft Remote Procedure Call (RPC) makes it easy to cross process and machine boundaries and carry data around. This network programming standard is one reason that networking with Microsoft Windows is so powerful.

However, because RPC hides network calls from individual processes, it obscures the details of the interactions between the computers. This can make it hard to be sure why threads are doing what they are doing -- or fail to do what they are supposed to do. As a result, debugging and troubleshooting RPC errors can be difficult. In addition, the vast majority of problems that appear to be RPC errors are actually configuration issues, or network connectivity issues, or other component issues.

Debugging Tools for Windows contains a tool called DbgRpc, as well as RPC-related debugger extensions. These can be used to analyze a variety of RPC problems on Windows XP and later versions of Windows.

These Windows versions can be configured to save RPC run-time state information. Different amounts of state information can be saved; this allows you to obtain the information you need without placing a significant burden on your computer. See Enabling RPC State Information for details.

This information can then be accessed through either the debugger or the DbgRpc tool. In each case, a collection of queries is available. See Displaying RPC State Information for details.

In many cases, you can troubleshoot a problem by using the techniques outlined in Common RPC Debugging Techniques.

If you want to explore the mechanics of how this information is stored, or if you want to devise your own techniques for state information analysis, see RPC State Information Internals.

Enabling RPC State Information

6/19/2019 • 2 minutes to read • Edit Online

Two different levels of RPC run-time state information can be gathered: **Server** information and **Full** information. This information gathering must be enabled before the debugger or DbgRpc can be used to analyze state information.

Only Windows XP and later versions of Windows support the gathering of RPC state information.

Gathering **Server** state information is very lightweight. It costs about 100 machine instructions per RPC call, resulting in no detectable load, even during performance tests. However, gathering this information does use memory (about 4KB per RPC server), so it is not recommended on a machine that is already experiencing memory pressure. **Server** information includes data about endpoints, threads, connection objects, and Server Call (SCALL) objects. This is sufficient to debug most RPC problems.

Gathering **Full** state information is more heavyweight. It includes all the information gathered at the **Server** level and, in addition, includes Client Call (CCALL) objects. **Full** state information is usually not needed.

To enable state information to be gathered on an individual machine, run the Group Policy Editor (Gpedit.msc). Under the Local Computer Policy, navigate to **Computer Configuration/Administrative Templates/System/Remote Procedure Call**. Under this node you will see the **Maintain RPC Troubleshooting State Information** item. When you edit its properties, you will see five possible states:

None
No state information will be maintained. Unless your machine is experiencing memory pressure, this is not recommended.

Server
Server state information will be gathered. This is the recommended setting on a single computer.

Full
Full state information will be gathered.

Auto1
On a computer with less than 64 MB of RAM, this is the same as **None**. On a computer with at least 64 MB of RAM, this is the same as **Server**.

Auto2
On a computer running Windows Server 2003 with less than 128 MB of RAM, or on any Windows XP computer, this is the same as **None**. On a Windows Server 2003 computer with at least 128 MB RAM, this is the same as **Server**.

This is the default.

If you want to simultaneously set these levels on a set of networked computers, use the Group Policy Editor to roll out a machine policy to the preferred set of machines. The policy engine will take care that the settings you want are propagated to the preferred set of machines. The **Auto1** and **Auto2** levels are especially useful in this case, because the operating system and amount of RAM on each computer may vary.

If the network includes computers running versions of Windows that are earlier than Windows XP, the settings will be ignored on those machines.

Displaying RPC State Information

12/5/2018 • 2 minutes to read • Edit Online

All RPC run-time state information is contained in cells. A cell is the smallest unit of information that can be viewed and updated individually. Both the DbgRpc tool and the RPC debugger extensions allow you to view the contents of any given cell or to run high-level queries.

Each key object in the RPC Run-Time will maintain one or more cells of information about its state. Each cell has a cell ID. When an object refers to another object, it does so by specifying that object's cell ID.

The key objects that the RPC Run-Time can maintain information about are endpoints, threads, connection objects, Server Call (SCALL) objects, and Client Call (CCALL) objects. Server Call objects are usually referred to simply as *call objects*.

The RPC state information queries produce the same information whether you are using the DbgRpc tool or the RPC debugger extensions. The following sections describe how queries are used in each vehicle:

Using the RPC Debugger Extensions

Using the DbgRpc Tool

The most basic query simply displays an individual cell:

Get RPC Cell Information

The following high-level queries are also available:

Get RPC Endpoint Information

Get RPC Thread Information

Get RPC Call Information

Get RPC Client Call Information

Using the RPC Debugger Extensions

12/5/2018 • 2 minutes to read • Edit Online

A variety of RPC debugger extensions are exported from Rpcexts.dll.

The RPC extensions used to display RPC state information will only run in user mode. They can be used from CDB (or NTSD) or from user-mode WinDbg.

The user-mode debugger must have a target application, but the target is irrelevant to the RPC extensions. If the debugger is not already running, you can simply start it with an uninteresting target (for example, **windbg notepad** or **cdb winmine**). Then, use **CTRL+C** in CDB or Debug | Break in WinDbg to stop the target and access the Debugger Command window.

If you need to analyze RPC state information from a remote computer, you should start the user-mode debugger on the computer that needs to be analyzed, and then use Remote Debugging.

Accessing RPC state information through the debugger is especially useful in stress environments, or when a debugger already happens to be running.

Using the DbgRpc Tool

12/5/2018 • 2 minutes to read • Edit Online

The DbgRpc tool (Dbgrpc.exe) is located in the root directory of the Debugging Tools for Windows installation and must be started in a Command Prompt window. Double-clicking the icon will not start this tool.

The Command Prompt window must be running under an account with administrative privileges on the local computer, or with domain administrative privileges.

DbgRpc makes no calls to any system services (such as LSASS). This makes it useful for debugging even if a system service has crashed, as long as the kernel is still running.

Using DbgRpc on a Remote Computer

DbgRpc can also be used to examine information from a remote machine. For this to work, the remote machine must be able to accept remote connections and authenticate remote users. If the remote machine's RPCSS (RPC Endpoint Mapper) service has crashed, DbgRpc will not be able to work. Administrative or domain administrative privileges on the remote machine are required.

The **-s** command-line option is used to specify the server name, and the **-p** parameter is used to specify the transport protocol. Both TCP and named pipe protocols are available. TCP is the recommended protocol; it should work in almost every situation.

Here is an example:

```
G:\>dbgrpc -s MyServer -p ncacn_ip_tcp -l -P 1e8 -L 0.1
Getting remote cell info ...
Endpoint
Status: Active
Protocol Sequence: LRPC
Endpoint name: OLE18
```

DbgRpc Command Line

For a description of the full command syntax, see **DbgRpc Command-Line Options**.

Get RPC Cell Information

12/5/2018 • 2 minutes to read • Edit Online

Detailed cell information is displayed by the **!rpcexts.getdbgcell** extension, or by DbgRpc when the **-l** switch is used.

The process ID of the process that contains the preferred cell must be specified, as well as the cell number.

In the following example, the process ID is 0x278, and the cell number is 0000.0002:

```
D:\wmsg>dbgrpc -l -P 278 -L 0.2
Getting cell info ...
Thread
Status: Dispatched
Thread ID: 0x1A4 (420)
Last update time (in seconds since boot):470.25 (0x1D6.19)
```

For details on the optional parameters, see **DbgRpc Command-Line Options**.

For a similar example using the RPC debugger extensions, see **!rpcexts.getdbgcell**.

Get RPC Endpoint Information

12/5/2018 • 2 minutes to read • Edit Online

Endpoint information is displayed by the **!rpcexts.getendpointinfo** extension, or by DbgRpc when the **-e** switch is used.

If an endpoint number is specified, information about that endpoint is shown. If it is omitted, the endpoints for all processes on the system are displayed.

The following example displays all endpoints. This is often a useful way to obtain process IDs and cell numbers that can be used as arguments for additional commands:

```
D:\wmsg>dbgrpc -e
Searching for endpoint info ...
## PID  CELL ID   ST PROTSEQ        ENDPOINT
-------------------------------------------------------
00a8 0000.0001 01            NMP \PIPE\InitShutdown
00a8 0000.0003 01            NMP \PIPE\SfcApi
00a8 0000.0004 01            NMP \PIPE\ProfMapApi
00a8 0000.0007 01            NMP \pipe\winlogonrpc
00a8 0000.0008 01           LRPC OLE5
00c4 0000.0001 01           LRPC ntsvcs
00c4 0000.0003 01            NMP \PIPE\ntsvcs
00c4 0000.0008 01            NMP \PIPE\scerpc
00d0 0000.0001 01            NMP \PIPE\lsass
00d0 0000.0004 01            NMP \pipe\WMIEP_d0
00d0 0000.000b 01            NMP \PIPE\POLICYAGENT
00d0 0000.000c 01           LRPC policyagent
0170 0000.0001 01           LRPC epmapper
0170 0000.0003 01            TCP 135
0170 0000.0005 01            SPX 34280
0170 0000.0006 01             NB 135
0170 0000.0007 01             NB 135
0170 0000.000b 01            NMP \pipe\epmapper
01b8 0000.0001 01            NMP \pipe\spoolss
01b8 0000.0003 01           LRPC spoolss
01b8 0000.0007 01           LRPC OLE7
00ec 0000.0001 01           LRPC OLE2
00ec 0000.0003 01           LRPC senssvc
00ec 0000.0007 01            NMP \pipe\tapsrv
00ec 0000.0008 01           LRPC tapsrvlpc
00ec 0000.000c 01            NMP \PIPE\ROUTER
00ec 0000.0010 01            NMP \pipe\WMIEP_ec
0214 0000.0001 01            NMP \PIPE\winreg
022c 0000.0001 01           LRPC LRPC0000022c.00000001
022c 0000.0003 01            TCP 1058
022c 0000.0005 01            SPX 24576
022c 0000.0006 01            NMP \PIPE\atsvc
02a8 0000.0001 01           LRPC OLE3
0370 0000.0001 01           LRPC OLE9
0278 0000.0001 01            TCP 1120
030c 0000.0001 01           LRPC OLE12
```

For details on the optional parameters, see **DbgRpc Command-Line Options**.

For a similar example using the RPC debugger extensions, see **!rpcexts.getendpointinfo**.

Get RPC Thread Information

12/5/2018 • 2 minutes to read • Edit Online

Thread information is displayed by the **!rpcexts.getthreadinfo** extension, or by DbgRpc when the **-t** switch is used.

The PID of a process must be specified. You may specify a thread within that process as well. If the thread is omitted, all threads within that process will be displayed.

In the following example, the process ID is 0x278 and the thread ID is omitted:

```
D:\wmsg>dbgrpc -t -P 278
Searching for thread info ...
## PID  CELL ID   ST TID      LASTTIME
---------------------------------
0278 0000.0002 01 000001a4 00072c09
0278 0000.0005 03 0000031c 00072bf5
```

For details on the optional parameters, see **DbgRpc Command-Line Options**.

For a similar example using the RPC debugger extensions, see **!rpcexts.getthreadinfo**.

Get RPC Call Information

12/5/2018 • 2 minutes to read • Edit Online

Server-side call (SCALL) information is displayed by the **!rpcexts.getcallinfo** extension, or by DbgRpc when the **-c** switch is used.

Four optional parameters are permitted. Three of these -- *CallID*, *IfStart*, and *ProcNum* -- are identifying information used by RPC to keep track of its calls. The fourth parameter, *ProcessID*, is the PID of the server process that owns the call. You should supply whatever parameters you know to narrow down the search.

If no parameters are supplied, all known SCALLs in the system will be displayed. The following is an example of this long display:

```
D:\wmsg>dbgrpc -c
Searching for call info ...
## PID  CELL ID   ST PNO IFSTART  TIDNUMBER CALLFLAG CALLID   LASTTIME CONN/CLN
-------------------------------------------------------------------------------
00c4 0000.0002 00 00f 82273fdc 0000.0007 00000001 00000002 0003595d 0000.0010
00c4 0000.0006 00 009 367abb81 0000.0015 00000001 0000004d 000185bd 0000.0005
00c4 0000.000a 00 007 367abb81 0000.002d 00000001 0000009f 00014672 0000.0009
00c4 0000.000c 00 007 367abb81 0000.002d 00000001 00000083 000122e3 0000.000b
00c4 0000.000d 00 03b 8d9f4e40 0000.002d 00000001 000000f7 0001aba5 0000.0020
00c4 0000.000e 00 03b 8d9f4e40 0000.0026 00000001 00000002 00023056 0000.0021
00c4 0000.000f 00 008 82273fdc 0000.001e 00000009 baadf00d 000366b4 00ec.03bc
00c4 0000.0012 00 00d 8d9f4e40 0000.0004 00000001 00000051 0000a334 0000.0011
00c4 0000.0014 00 000 367abb81 0000.0015 00000001 0000004c 0002db53 0000.0013
00c4 0000.0017 00 007 367abb81 0000.0015 00000001 00000006 0000d102 0000.0016
00c4 0000.0019 00 007 367abb81 0000.0004 00000001 00000006 0000f09e 0000.0018
00c4 0000.001b 00 009 65a93890 0000.0007 00000001 0000012e 00630f65 0000.001a
00c4 0000.001e 00 026 8d9f4e40 0000.0015 00000001 0000037d 0005e579 0000.002c
00c4 0000.001f 00 008 82273fdc 0000.0033 00000009 baadf00d 000145b3 00c4.02f8
00c4 0000.0023 00 000 367abb81 0000.0004 00000001 0000007e 000372f3 0000.0022
00c4 0000.0025 00 03b 8d9f4e40 0000.0026 00000001 0000000b 000122e3 0000.0024
00c4 0000.0027 00 000 367abb81 0000.002d 00000001 0000000b 00012e27 0000.0028
00c4 0000.002a 00 008 82273fdc 0000.0033 00000009 baadf00d 0001245f 022c.0290
00c4 0000.002f 00 007 367abb81 0000.0026 00000001 0000000a 0002983c 0000.002e
00c4 0000.0031 00 004 3ba0ffc0 0000.0026 00000001 00000007 0005c439 0000.001c
00c4 0000.0032 00 00b 82273fdc 0000.0039 00000009 baadf00d 00687db6 00d0.01d4
00c4 0000.0036 00 007 367abb81 0000.0030 00000001 00000065 0003a5e1 0000.0035
00c4 0000.0037 00 00d 8d9f4e40 0000.0015 00000001 0000033f 000376fa 0000.002b
00c4 0000.0038 00 008 8d9f4e40 0000.0015 00000001 00000803 0018485c 0000.003b
00c4 0000.003c 00 00b 82273fdc 0000.0034 00000009 baadf00d 0001f956 00a8.0244
00c4 0000.003d 00 008 82273fdc 0000.0034 00000009 baadf00d 0001ff02 01b8.037c
0170 0000.0009 00 002 e60c73e6 0000.0013 00000009 baadf00d 0005a371 00ec.031c
0170 0000.000a 00 002 0b0a6584 0000.0002 00000009 baadf00d 000126ae 00c4.0130
0170 0000.000c 00 002 0b0a6584 0000.0010 00000009 baadf00d 00012bc4 022c.0290
0170 0000.000d 00 003 00000136 0000.001b 00000009 baadf00d 0005ba71 00ec.0310
0170 0000.000e 00 000 412f241e 0000.0002 00000009 baadf00d 00012f21 02a8.029c
0170 0000.0010 00 003 00000136 0000.0013 00000009 00000003 000341da 0370.0060
0170 0000.0011 00 006 e60c73e6 0000.001b 00000009 baadf00d 000f1d00 0370.0328
0170 0000.0017 00 002 0b0a6584 0000.001b 00000009 baadf00d 0006c803 0278.0184
0170 0000.001a 00 004 00000136 0000.0012 00000001 baadf00d 00038e9b 00ec.0348
00ec 0000.0006 00 009 00000134 0000.0011 00000009 baadf00d 000b233f 0170.0244
00ec 0000.000b 00 001 2f5f6520 0000.001c 00000009 baadf00d 00035510 00ec.0334
00ec 0000.000e 00 001 629b9f66 0000.0014 00000009 baadf00d 00035813 00ec.01c4
00ec 0000.0012 00 000 629b9f66 0000.0014 00000009 baadf00d 00026cc6 00a8.0164
00ec 0000.001b 00 001 2f5f6520 0000.0004 00000001 baadf00d 000352c1 00ec.03a8
02a8 0000.0004 00 009 00000134 0000.0002 00000009 baadf00d 0009a540 0170.0244
0370 0000.0006 00 003 00000134 0000.0005 0000000b baadf00d 0002e7cd 00ec.0350
0370 0000.0008 00 009 00000134 0000.0007 0000000b 01cee9e4 000838fa 0170.0244
0278 0000.0004 02 000 19bb5061 0000.0002 00000001 00000001 00072c09 0000.0003
```

For details on the optional parameters, see **DbgRpc Command-Line Options**.

For a similar example using the RPC debugger extensions, see **!rpcexts.getcallinfo**.

Get RPC Client Call Information

12/5/2018 • 2 minutes to read • Edit Online

Client call (CCALL) call information is displayed by the **!rpcexts.getclientcallinfo** extension, or by DbgRpc when the **-a** switch is used.

Four optional parameters are permitted. Three of these -- *CallID*, *IfStart*, and *ProcNum* -- are identifying information used by RPC to keep track of its calls. The fourth parameter, *ProcessID*, is the PID of the process that owns the call. You should supply whatever parameters you know to narrow down the search.

If no parameters are supplied, all known CCALLs in the system will be displayed. The following is an example of this (potentially long) display:

```
D:\wmsg>dbgrpc -a
Searching for call info ...
## PID  CELL ID   PNO  IFSTART  TIDNUMBER CALLID   LASTTIME PS CLTNUMBER ENDPOINT
--------------------------------------------------------------------------------
0390 0000.0001 0000 19bb5061 0000.0000 00000001 00072bff 07 0000.0002 1120
```

For details on the optional parameters, see **DbgRpc Command-Line Options**.

For a similar example using the RPC debugger extensions, see **!rpcexts.getclientcallinfo**.

Note Information about Client Call objects is only gathered if the **Full** state information is being gathered. See Enabling RPC State Information for details.

Common RPC Debugging Techniques

12/5/2018 • 2 minutes to read • Edit Online

This section describes four common RPC-related problems. RPC state information can be used to troubleshoot these problems.

Either the DbgRpc tool or the RPC debugger extensions can be used in any of these four situations.

Analyzing a Stuck Call Problem

Tracking Contention in the Server Process

Checking for Stuck Threads

Identifying the Caller From the Server Thread

Analyzing a Stuck Call Problem

12/5/2018 • 3 minutes to read • Edit Online

A common problem occurs when a process makes an RPC call, directly or indirectly, while holding a critical section or a resource. In this case, the RPC call goes to another process or machine and dispatches to the manager routine (server routine), which then hangs or takes too long. This causes the original caller to encounter a critical section time-out.

When examined through the debugger, RPC is on top of the stack of the thread owning the critical section, but it is not clear what it is waiting for.

Here is one example of such a stack. Many variations are possible.

```
0:002> ~1k
ChildEBP RetAddr
0068fba0 77e9e8eb ntdll!ZwWaitForSingleObject+0xb
0068fbc8 4efeff73 KERNEL32!WaitForSingleObjectEx+0x5a
0068fbe8 4eff0012 RPCRT4!UTIL_WaitForSyncIO+0x21
0068fc0c 4efe6e2b RPCRT4!UTIL_GetOverlappedResultEx+0x44
0068fc44 4ef973bf RPCRT4!WS_SyncRecv+0x12a
0068fc68 4ef98d5a RPCRT4!OSF_CCONNECTION__TransSendReceive+0xcb
0068fce4 4ef9b682 RPCRT4!OSF_CCONNECTION__SendFragment+0x297
0068fd38 4ef9a5a8 RPCRT4!OSF_CCALL__SendNextFragment+0x272
0068fd88 4ef9a9cb RPCRT4!OSF_CCALL__FastSendReceive+0x165
0068fda8 4ef9a7f8 RPCRT4!OSF_CCALL__SendReceiveHelper+0xed
0068fdd4 4ef946a7 RPCRT4!OSF_CCALL__SendReceive+0x37
0068fdf0 4efd56b3 RPCRT4!I_RpcSendReceive+0xc4
0068fe08 01002850 RPCRT4!NdrSendReceive+0x4f
0068ff40 01001f32 rtclnt+0x2850
0068ffb4 77e92ca8 rtclnt+0x1f32
0068ffec 00000000 KERNEL32!CreateFileA+0x11b
```

Here's how to troubleshoot this problem.

Troubleshooting a stuck call problem

1. Make sure the debugger is debugging the process that owns the stuck cell. (This is the process containing the client thread that is suspected of hanging in RPC.)

2. Get the stack pointer of this thread. The stack will look like the one shown in the preceding example. In this example, the stack pointer is 0x0068FBA0.

3. Get the call information for this thread. In order to do that, use the **!rpcexts.rpcreadstack** extension with the thread stack pointer as its parameter, as follows:

   ```
   0:001> !rpcexts.rpcreadstack 68fba0
   CallID: 1
   IfStart: 19bb5061
   ProcNum: 0
           Protocol Sequence:      "ncacn_ip_tcp"  (Address: 00692ED8)
           NetworkAddress: ""      (Address: 00692F38)
           Endpoint:       "1120"  (Address: 00693988)
   ```

   The information displayed here will allow you to trace the call.

4. The network address is empty, which indicates the local machine. The endpoint is 1120. You need to

determine which process hosts this endpoint. This can be done by passing this endpoint number to the **!rpcexts.getendpointinfo** extension, as follows:

```
0:001> !rpcexts.getendpointinfo 1120
Searching for endpoint info ...
PID  CELL ID   ST PROTSEQ        ENDPOINT
-----------------------------------------
0278 0000.0001 01            TCP 1120
```

5. From the preceding information, you can see that process 0x278 contains this endpoint. You can determine if this process knows anything about this call by using the **!rpcexts.getcallinfo** extension. This extension needs four parameters: *CallID*, *IfStart*, and *ProcNum* (which were found in step 3), and the *ProcessID* of 0x278:

```
0:001> !rpcexts.getcallinfo 1 19bb5061 0 278
Searching for call info ...
PID  CELL ID   ST PNO IFSTART  TIDNUMBER CALLFLAG CALLID   LASTTIME CONN/CLN
-------------------------------------------------------------------------
0278 0000.0004 02 000 19bb5061 0000.0002 00000001 00000001 00072c09 0000.0003
```

6. The information in step 5 is useful, but somewhat abbreviated. The cell ID is given in the second column as 0000.0004. If you pass the process ID and this cell number to the **!rpcexts.getdbgcell** extension, you will see a more readable display of this cell:

```
0:001> !rpcexts.getdbgcell 278 0.4
Getting cell info ...
Call
Status: Dispatched
Procedure Number: 0
Interface UUID start (first DWORD only): 19BB5061
Call ID: 0x1 (1)
Servicing thread identifier: 0x0.2
Call Flags: cached
Last update time (in seconds since boot):470.25 (0x1D6.19)
Owning connection identifier: 0x0.3
```

This shows that the call is in state "dispatched", which means is has left the RPC Run-Time. The last update time is 470.25. You can learn the current time by using the **!rpcexts.rpctime** extension:

```
0:001> !rpcexts.rpctime
Current time is: 6003, 422
```

This shows that the last contact with this call was approximately 5533 seconds ago, which is about 92 minutes. Thus, this must be a stuck call.

7. As a last step before you attach a debugger to the server process, you can isolate the thread that should currently service the call by using the Servicing thread identifier. This is another cell number; it appeared in step 6 as "0x0.2". You can use it as follows:

```
0:001> !rpcexts.getdbgcell 278 0.2
Getting cell info ...
Thread
Status: Dispatched
Thread ID: 0x1A4 (420)
Last update time (in seconds since boot):470.25 (0x1D6.19)
```

Now you know that you are looking for thread 0x1A4 in process 0x278.

It is possible that the thread was making another RPC call. If necessary, you can trace this call by repeating this procedure.

Note This procedure shows how to find the server thread if you know the client thread. For an example of the reverse technique, see Identifying the Caller From the Server Thread.

Tracking Contention in the Server Process

12/5/2018 • 2 minutes to read • Edit Online

In order to service incoming requests, RPC will maintain a set of worker threads. Ideally, the number of threads will be small. However, this ideal situation has only been seen in lab environments, where the server manager routines are carefully tuned. In a real situation, the number of threads will vary depending on server workload, but it can be anywhere from 1 to 50.

If the number of worker threads is above 50, you may have excessive contention in the server process. Common causes of this are indiscriminate use of the heap, memory pressure, or serializing most activities in a server through a single critical section.

To see the number of threads in a given server process, use the **!rpcexts.getthreadinfo** extension, or use DbgRpc with the **-t** switch. Supply the process ID (in the following example, 0xC4):

```
D:\wmsg>dbgrpc -t -P c4
Searching for thread info ...
## PID  CELL ID   ST TID      LASTTIME
---------------------------------
00c4 0000.0004 03 0000011c 000f164f
00c4 0000.0007 03 00000120 008a6290
00c4 0000.0015 03 0000018c 008a6236
00c4 0000.0026 03 00000264 0005c443
00c4 0000.002d 03 00000268 000265bb
00c4 0000.0030 03 0000026c 000f1d32
00c4 0000.0034 03 00000388 007251e9
```

In this case, there are only seven worker threads, which is reasonable.

If there are over 100 threads, a debugger should be attached to this process and the cause investigated.

Note Running queries such as **dbgrpc -t** remotely is expensive to the server and the network. If you use this query in a script, you should make sure this command is not run too often.

Checking for Stuck Threads

12/5/2018 • 2 minutes to read • Edit Online

RPC needs its worker threads available in order to perform normally. A common problem is that some component in the same process will deadlock while holding one of the global critical sections (for example, loader lock or heap lock). This will cause many threads to hang -- very possibly including some RPC worker threads.

If this occurs, the RPC server will not respond to the outside world. RPC calls to it will return RPC_S_SERVER_UNAVAILABLE or RPC_S_SERVER_TOO_BUSY.

A similar problem can result if a faulty driver prevents IRPs from completing and reaching the RPC server.

If you suspect that one of these problems may be occurring, use DbgRpc with the **-t** switch (or use the **!rpcexts.getthreadinfo** extension). The process ID should be used as a parameter. In the following example, assume the process ID is 0xC4:

```
D:\wmsg>dbgrpc -t -P c4
Searching for thread info ...
## PID  CELL ID   ST TID      LASTTIME
----------------------------------
00c4 0000.0004 03 0000011c 000f164f
00c4 0000.0007 03 00000120 008a6290
00c4 0000.0015 03 0000018c 008a6236
00c4 0000.0026 03 00000264 0005c443
00c4 0000.002d 03 00000268 000265bb
00c4 0000.0030 03 0000026c 000f1d32
00c4 0000.0034 03 00000388 007251e9
```

The TID column gives the thread ID for each thread. The LASTTIME column contains the time stamp of the last change in state for each thread.

Whenever the server receives a request, at least one thread will change state, and its time stamp will be updated. Therefore, if an RPC request is made to the server and the request fails but none of the time stamps change, this indicates that the request is not actually reaching the RPC Run-Time. You should investigate the cause of this.

Identifying the Caller From the Server Thread

12/5/2018 • 2 minutes to read • Edit Online

It is possible to determine what made a given RPC call, even if the only information you have is the server thread that serviced the call.

This can be very useful -- for example, to find out who passed invalid parameters to an RPC call.

Depending on which protocol sequence is used by this particular call, you can get varying degrees of detail. Some protocols (such as NetBios) do not have this information at all.

Identifying the caller from the server thread

1. Start a user-mode debugger with the server thread as the target.
2. Get the process ID by using the **| (Process Status)** command:

```
0:001> |
  0     id: 3d4 name: rtsvr.exe
```

3. Get the active calls in this process by using the **!rpcexts.getcallinfo** extension. (See the reference page for an explanation of the syntax.) You need to supply the process ID of 0x3D4:

```
0:001> !rpcexts.getcallinfo 0 0 FFFF 3d4
Searching for call info ...
PID  CELL ID   ST PNO IFSTART  THRDCELL  CALLFLAG CALLID   LASTTIME CONN/CLN
-------------------------------------------------------------------------
03d4 0000.0004 02 000 19bb5061 0000.0002 00000001 00000001 00a1aced 0000.0003
```

   Look for calls with status 02 or 01 (dispatched or active). In this example, the process only has one call. If there were more, you would have to use the **!rpcexts.getdbgcell** extension with the cell number in the THRDCELL column. This would allow you to examine the thread IDs so you could determine which call you were interested in.

4. After you know which call you are interested in, look at the cell number in the CONN/CLN column. This is the cell ID of the connection object. In this case, the cell number is 0000.0003. Pass this cell number and the process ID to **!rpcexts.getdbgcell**:

```
0:001> !rpcexts.getdbgcell 3d4 0.3
Getting cell info ...
Connection
Connection flags: Exclusive
Authentication Level: Default
Authentication Service: None
Last Transmit Fragment Size: 24 (0x6F56D)
Endpoint for the connection: 0x0.1
Last send time (in seconds since boot):10595.565 (0x2963.235)
Last receive time (in seconds since boot):10595.565 (0x2963.235)
Getting endpoint info ...
Process object for caller is 0xFF9DF5F0
```

This extension will display all the information available about the client of this connection. The amount of actual information will vary, depending on the transport being used.

In this example, local named pipes are being used as the transport and the process object address of the caller is displayed. If you attach a kernel debugger (or start a local kernel debugger), you can use the **!process** extension to interpret this process address.

If LRPC is used as the transport, the process ID and thread ID of the caller will be displayed.

If TCP is used as the transport, the IP address of the caller will be displayed.

If remote named pipes are used as the transport, no information will be available.

Note The previous example shows how to find the client thread if you know the server thread. For an example of the reverse technique, see Analyzing a Stuck Call Problem.

RPC State Information Internals

12/5/2018 • 5 minutes to read • Edit Online

This section provides details of the internal structure of the state information gathered by the RPC Run-Time.

All RPC run-time state information is contained in cells. A cell is the smallest unit of information that can be viewed and updated individually.

Each key object in the RPC Run-Time will maintain one or more cells of information about its state. Each cell has a cell ID. When an objects refers to another object, it does so by specifying that object's cell ID. The key objects that the RPC Run-Time can maintain information about are endpoints, threads, connection objects, Server Call (SCALL) objects, and Client Call (CCALL) objects.

When an RPC server is running, the RPC Run-Time listens on a set of endpoints using one or more worker threads. Whenever data is transmitted to the server, a thread picks up the data and determines what the incoming request is. If the request is to create a connection, a Connection object is created, and this object then services all calls on the connection. When an RPC call is made on the connection, the Connection object instantiates a Server Call (SCALL) object corresponding to the Client Call (CCALL) object. This Server Call object then handles this particular call.

When an RPC client is running, the RPC Run-Time creates a Client Call object each time a call is made. This Client Call object contains information about this particular call.

Endpoint Cells

From the RPC run-time's point of view, an endpoint is an entry point through which the particular server can be contacted. The endpoint is always associated with a given RPC transport. The endpoint state information is used to associate a client call with a particular process on the server.

The fields in an endpoint cell are:

ProtseqType
The protocol sequence for this endpoint.

Status
The status value: *allocated*, *active*, or *inactive*. Most endpoints are active. An endpoint has *allocated* status when the creation process has started, but is not complete yet. An endpoint is *inactive* if it is no longer in use (for example, when a protocol has been uninstalled).

EndpointName
The first 28 characters of the endpoint name.

Thread Cells

Server threads are worker threads (standard Win32 threads for use by RPC).

The fields in a thread cell are:

Status
The status value: *processing*, *dispatched*, *allocated*, or *idle*. A *processing* thread is one that is within the Run-Time and is processing information. A *dispatched* thread has already dispatched (called) to the server-provided manager routine (usually just called the *server routine*). An *allocated* thread has been cached. An *idle* thread is available to service requests.

LastUpdateTime
The time (in milliseconds after boot) when the information was last updated.

TID
The thread ID of this thread. This is useful when trying to correlate with the thread list in the debugger.

Connection Object Cells

The fields in a connection object cell are:

Flags
Flag values include *exclusive/non-exclusive*, *authentication level*, and *authentication service*.

LastTransmitFragmentSize
The size of the last fragment transmitted over the connection.

Endpoint
The cell ID of the endpoint that this connection was picked up from.

LastSendTime
The last time data was sent on a connection.

LastReceiveTime
The last time data was received on a connection.

Server Call Object Cells

The fields in a Server Call (SCALL) object cell are:

Status
The status value: *allocated*, *active*, or *dispatched*. An *allocated* call is inactive and cached. When a call is *active*, the RPC Run-Time is processing information related to this call. When a call is *dispatched*, the manager routine (server routine) has been called and has not returned yet.

ProcNum
The procedure number (operation number, in netmon capture files) of this call. The RPC Run-Time identifies individual routines from an interface by numbering them by position in the IDL file. The first routine in the interface will be number zero, the second number one, and so on.

InterfaceUUIDStart
The first DWORD of the interface UUID.

ServicingTID
The cell ID of the thread that is servicing this call. If the call is not *active* or *dispatched*, this contains stale information.

CallFlags
These flag values indicate whether this is the cached call in an exclusive connection, whether this is an asynchronous call, whether this is a pipe call, and whether this is an LRPC or OSF call.

LastUpdateTime
The time (in milliseconds after boot) when the call object state information was last updated.

PID
The Process ID of the caller. Valid only for LRPC calls.

TID
The Thread ID of the caller. Valid only for LRPC calls.

Client Call Object Cells

A Client Call (CCALL) object is broken into two cells, because the information about a client call is too large to fit in one cell. The first cell is called *Client Call Information*, and the second is called *Call Target Information*. Most tools will show the information together, so you do not need to distinguish between them.

Information about client calls is not maintained unless you are gathering Full state information. There is one exception to this rule: information about client calls made within a server call is maintained even when only Server state information is being gathered. This allows you to trace calls spanning multiple hops.

The fields in the Client Call Information cell are:

ProcNum
The procedure number (operation number, in netmon capture files) of the method being called. The RPC Run-Time identifies individual routines from an interface by numbering them by position in the IDL file. The first routine in the interface will be number zero, the second number one, and so on.

ServicingThread
The cell ID of the thread on which this call is made.

IfStart
The first DWORD of the interface UUID on which the call is made.

Endpoint
The first 12 characters of the endpoint on the server to which the call was made.

The fields in the Call Target Information cell are:

ProtocolSequence
The protocol sequence for this call.

LastUpdateTime
The time (in milliseconds after boot) when the information about the client call or the call target was updated.

TargetServer
The first 24 characters of the name of the server to which the call is made.

ACPI Debugging

12/5/2018 • 2 minutes to read • Edit Online

This section includes:

The AMLI Debugger

Other ACPI Debugging Extensions

The AMLI Debugger

12/5/2018 • 2 minutes to read • Edit Online

This section includes:

Introduction to the AMLI Debugger

Setting Up an AML Debugging Session

Basic AML Debugging

Using AMLI Debugger Extensions

Using AMLI Debugger Commands

AML Debugging Examples

Introduction to the AMLI Debugger

12/5/2018 • 2 minutes to read • Edit Online

There are significant differences between debugging standard kernel-mode code and debugging an ACPI (Advanced Configuration and Power Interface) BIOS.

Whereas Windows and its drivers are composed of binary machine code compiled for a specific processor, the core of an ACPI BIOS is not in machine code. Rather, it is stored as ACPI Machine Language (AML) and is processed by the Microsoft AML interpreter as it is run.

The Microsoft AMLI Debugger is a set of special debugging tool that can debug AML code.

In versions of Windows before Windows 10, version 1803 checked builds of the Windows ACPI driver (Acpi.sys) were used. This no longer the case.

The AMLI Debugger is completely 64-bit aware. No matter what processor is being used by the target computer or the host computer, the AMLI Debugger will function correctly.

Setting Up an AML Debugging Session

12/5/2018 • 2 minutes to read • Edit Online

The AMLI Debugger code is contained in Acpi.sys. In order to fully perform AML debugging, this driver must be installed on your target computer, which it normally is.

To activate breaking into debugger on free builds, use the **!amli set dbgbrkon** command, that is part of the AMLI Debugger extensions. For more information, see **!amli set**.

See Also

The AMLI Debugger

Basic AML Debugging

12/5/2018 • 2 minutes to read • Edit Online

The AMLI Debugger supports two types of specialized commands: *AMLI Debugger extensions* and *AMLI Debugger commands*.

When you are performing AML debugging, you should carefully distinguish between two different kinds of prompts that will appear in the Debugger Command window:

- When you see the **kd>** prompt, you are controlling the kernel debugger. All the standard kernel debugger commands and extensions are available. In addition, the AMLI Debugger extensions are also available. These extensions have a syntax of **!amli** *command*. The AMLI Debugger commands are not available in this mode.
- When you see the **AMLI(? for help)->** prompt, you are controlling the AMLI Debugger. (When you are using WinDbg, this prompt will appear in the top pane of the Debugger Command window, and an **Input>** prompt will appear in the bottom pane.) From this prompt, you can enter any AMLI Debugger command. You can also enter any AMLI Debugger extension; these extensions should not be prefixed with **!amli**. The standard kernel debugging commands are not available in this mode.
- When you see no prompt at all, the target computer is running.

At the beginning of any debugging session, you should set your AMLI Debugger options with the **!amli set** extension. The **verboseon**, **traceon**, and **errbrkon** options are also very useful. You should consider activating the **spewon** option. See the extension reference page for details.

There are several ways for the AMLI Debugger to become active:

- If a breakpoint in AML code is encountered, ACPI will break into the AMLI Debugger.
- If a serious error or exception occurs within AML code (such as an **int 3**), ACPI will break into the AMLI Debugger.
- If the **errbrkon** option has been set, any AML error will cause ACPI to break into the AMLI Debugger.
- If you want to deliberately break into the AMLI Debugger, use the **!amli debugger** extension and then the **g (Go)** command. The next time any AML code is executed by the interpreter, the AMLI Debugger will take over.

When you are at the AMLI Debugger prompt, you can type **q** to return to the kernel debugger, or type **g** to resume normal execution.

The following extensions are especially useful for AML debugging:

- The **!amli dns** extension displays the ACPI namespace for a particular object, the namespace tree subordinate to that object, or even the entire namespace tree. This command is especially useful in determining what a particular namespace object is -- whether it is a method, a fieldunit, a device, or another type of object.
- The **!amli find** extension takes the name of any namespace object and returns its full path.
- The **!amli u** extension unassembles AML code.
- The **!amli lc** extension displays brief information about all active ACPI contexts.
- The **!amli r** extension displays detailed information about the current context of the interpreter. This is

useful when the AMLI Debugger prompt appears after an error is detected.

- Breakpoints can be set and controlled within AML code. Use **!amli bp** to set a breakpoint, **!amli bc** to clear a breakpoint, **!amli bd** to disable a breakpoint, **!amli be** to re-enable a breakpoint, and **!amli bl** to list all breakpoints.
- The AMLI Debugger is able to run, step, and trace through AML code. Use the **run**, **p**, and **t** commands to perform these actions.

For a full list of extensions and commands, see Using AMLI Debugger Extensions and Using AMLI Debugger Commands.

See Also

The AMLI Debugger

Using AMLI Debugger Extensions

12/5/2018 • 2 minutes to read • Edit Online

The AMLI Debugger extension commands are contained in the extension module Kdexts.dll and use the following syntax:

```
kd> !amli command [parameters]
```

As with any extension module, after it has been loaded you can omit the **acpikd** prefix.

If you are at the AMLI Debugger prompt, you can execute any of these extension commands by simply entering the *command* name without the **!amli** prefix:

```
AMLI(? for help)-> command [parameters]
```

When you are at this prompt, the **!amli debugger** command is not available (because it would be meaningless). Also, the help command (**?**) at this prompt shows all AMLI Debugger extensions and commands, while the **!amli ?** extension only displays help on actual extensions.

ACTION	EXTENSION COMMAND
Display Help	**!amli ?**
Set AML Breakpoint	**!amli bp**
List AML Breakpoints	**!amli bl**
Disable AML Breakpoint	**!amli bd**
Enable AML Breakpoint	**!amli be**
Clear AML Breakpoint	**!amli bc**
Enter AMLI Debugger	**!amli debugger**
Display Event Log	**!amli dl**
Clear Event Log	**!amli cl**
Display Heap	**!amli dh**

ACTION	EXTENSION COMMAND
Display Data Object	**!amli do**
Display Stack	**!amli ds**
Display Namespace Object	**!amli dns**
Find Namespace Object	**!amli find**
Display Nearest Method	**!amli ln**
List All Contexts	**!amli lc**
Display Context Information	**!amli r**
Unassemble AML Code	**!amli u**
Set AMLI Debugger Options	**!amli set**

See Also

The AMLI Debugger

Using AMLI Debugger Commands

12/5/2018 • 4 minutes to read • Edit Online

The following commands can be issued from the AMLI Debugger prompt.

GENERAL CATEGORY	SPECIFIC ACTION	AMLI DEBUGGER COMMANDS
Controlling the Debugger	Continue Execution Break to Kernel Debugger	**g q**
Controlling AML Execution	Run Method Step Over AML Code Trace Into AML Code	**run p t**
Controlling Trace Mode Settings	Configure Trace Mode	**trace**
Notifying a Namespace Object	Notify Namespace Object	**notify**
Displaying the Object Count Table	Display Object Count Table	**dc**
Accessing Memory	Display Data Display Data Bytes Display Data Words Display Data DWORDs Display Data String Edit Memory	**d db dw dd da e**
Accessing Ports	Read Byte from Port Read Word from Port Read DWORD from Port Write Byte to Port Write Word to Port Write DWORD to Port	**i iw id o ow od**
Displaying Help	Display Help	**?**

Controlling the Debugger

These commands exit the AMLI Debugger. The **g** command will resume normal execution of the target computer, and the **q** command will freeze the target computer and break into the kernel debugger.

g

q

Controlling AML Execution

These commands allow you to run or step through AML methods. The **run** command begins execution at a specified point. The **p** and **t** commands allow you to step through one instruction at a time. If a function call is encountered, the **p** command treats the function as a single step, while the **t** command traces into the new function one instruction at a time.

run *MethodName* **[*ArgumentList*]**

run *CodeAddress* **[*ArgumentList*]**

p

t

MethodName
Specifies the full path and name of a method. Execution will start at the beginning of this method's memory location.

CodeAddress
Specifies the address where execution is to begin.

ArgumentList
Specifies a list of arguments to be passed to the method. Each argument must be an integer. Multiple arguments should be separated with spaces.

Controlling Trace Mode Settings

The **trace** command controls the AML interpreter's trace mode settings. If this command is used with no parameters, the current trace mode settings are displayed.

trace [trigon|trigoff] [level=*Level***] [add=***TPStrings***] [zap=***TPNumbers***]**

trigon
Activates trace trigger mode.

trigoff
Deactivates trace trigger mode.

Level
Specifies the new setting for the trace level.

TPStrings
Specifies one or more trigger points to be added. Each trigger point is specified by name. Multiple trigger point strings should be separated by commas.

TPNumbers
Specifies one or more trigger points to be deleted. Each trigger point is specified by number. Multiple trigger point numbers should be separated by commas. To see a list of trigger point numbers, use the **trace** command with no parameters.

Notifying a Namespace Object

The **notify** command sends a notification to an ACPI namespace object. The notification will be placed in the specified object's queue.

notify *ObjectName Value*

notify *ObjectAddress Value*

ObjectName
Specifies the full namespace path of the object to be notified.

ObjectAddress
Specifies the address of the object to be notified.

Value
Specifies the notification value.

Displaying the Object Count Table

The **dc** command displays the memory object count table.

dc

Accessing Memory

The memory access commands allow you to read and write to memory. When reading memory, you can choose the size of the memory units with the **db**, **dw**, **dd** or **da** command. A simple **d** command displays memory in the most recently-chosen units. If this is the first display command used, byte units are used.

If no address or method is specified, display will begin where the previous display command ended.

These commands have the same effect as the standard kernel debugger memory commands; they are duplicated within the AMLI Debugger for easy access.

d[b|w|d|a] [[l=*Length***] [** *Method* **| [%%]***Address* **]]**

e [%%]*Address Datalist*

b
Specifies that the data should be displayed in byte units.

w
Specifies that the data should be displayed in word (16-bit) units.

d
Specifies that the data should be displayed in DWORD (32-bit) units.

a
Specifies that the data should be displayed as a string. The data is displayed as ASCII characters. The display terminates when a NULL character is read, or when *Length* characters have been displayed.

Length
Specifies the number of bytes to be displayed. *Length* must be a hexadecimal number (without an **0x** prefix). If *Length* is omitted, the default display size is 0x80 bytes.

Method
Specifies the full path and name of a method. The display will start at the beginning of this method's memory location.

Address
Specifies the memory address where reading or writing will begin. If the address is prefixed with two percent signs (**%%**), it is interpreted as a physical address. Otherwise, it is interpreted as a virtual address.

DataList
Specifies the data to be written to memory. Each item in the list can be either a hexadecimal byte or a string. When a string is used, it must be enclosed in quotation marks. Multiple items should be separated by spaces.

Accessing Ports

The port commands allow you to send output or receive input from a data port. The **i** and **o** commands transfer single bytes, the **iw** and **ow** commands transfer words (16 bits), and the **id** and **od** commands transfer DWORDS (32 bits).

These commands have the same effect as the standard kernel debugger port commands; they are duplicated within the AMLI Debugger for easy access.

i *Port*

iw *Port*

id *Port*

o *Port DataForPort*

ow *Port DataForPort*

od *Port DataForPort*

Port
Specifies the address of the port to be accessed. The port size must match the command chosen.

DataForPort
Specifies the data to be written to the port. The size of this data must match the command chosen.

Displaying Help

This command displays help text for the AMLI Debugger commands.

? [*Command***]**

Command
Specifies the command for which to display help. If this is omitted, a list of all AMLI Debugger commands and AMLI Debugger extensions is displayed.

See Also

The AMLI Debugger

AML Debugging Examples

12/5/2018 • 3 minutes to read • Edit Online

Here are examples that illustrate how to get started with AML debugging.

Investigating a Frozen Computer

If the target computer has frozen and you suspect it may be an ACPI problem, begin by using the **!amli lc** extension to display all the active contexts:

```
kd> !amli lc
*Ctxt=ffffffff8128d000, ThID=ffffffff81277880, Flgs=----R----, pbOp=ffffffff8124206c,
Obj=\_SB.PCI0.ISA0.FDC0._CRS
```

If no contexts are displayed, the error is probably not ACPI-related.

If there are contexts shown, look for the one marked with an asterisk. This is the *current context* (the one that is being executed by the interpreter at the present moment).

In this example, the target computer is running Windows on a 32-bit processor. Therefore all addresses are cast to 64 bits, producing a gratuitous FFFFFFFF in the high 32 bits. The abbreviation **pbOp** indicates the instruction pointer ("pointer to binary op codes"). The **Obj** field gives the full path and name of the method as it appears in the ACPI tables. For a description of the flags, see **!amli lc**.

You can use the **!amli u** command to disassemble the _CRS method as follows:

```
kd> !amli u \_SB.PCI0.ISA0.FDC0._CRS

ffffffff80e4a535 : CreateDWordFieldCRES, 0x76, RAMT)
ffffffff80e4a540 : CreateDWordField(CRES, 0x82, PCIT)
ffffffff80e4a54b : Add(MLEN(), 0x100000, RAMT)
ffffffff80e4a559 : Subtract(0xffe00000, RAMT, PCIT)
ffffffff80e4a567 : Return(CRES)
```

Breaking Into the AMLI Debugger

The **!amli debugger** command causes the AML interpreter to break into the AMLI Debugger the next time any AML code is executed.

After the AMLI Debugger prompt appears, you can use any of the AMLI Debugger commands. You can also use **!amli** extension commands without prefixing them with "!amli":

```
kd> !amli debugger
kd> g

AMLI(? for help)-> find _crs
\_SB.LNKA._CRS
\_SB.LNKB._CRS
\_SB.LNKC._CRS
\_SB.LNKD._CRS
\_SB.PCI0._CRS
\_SB.PCI0.LPC.NCP._CRS
\_SB.PCI0.LPC.PIC._CRS
\_SB.PCI0.LPC.TIME._CRS
\_SB.PCI0.LPC.IDMA._CRS
\_SB.PCI0.LPC.RTC._CRS
\_SB.PCI0.LPC.SPKR._CRS
\_SB.PCI0.LPC.FHUB._CRS
\_SB.PCI0.SBD1._CRS
\_SB.PCI0.SBD2._CRS
\_SB.MBRD._CRS

AMLI(? for help)-> u \_SB.PCI0._CRS

ffffffff80e4a535 : CreateDWordFieldCRES, 0x76, RAMT)
ffffffff80e4a540 : CreateDWordField(CRES, 0x82, PCIT)
ffffffff80e4a54b : Add(MLEN(), 0x100000, RAMT)
ffffffff80e4a559 : Subtract(0xffe00000, RAMT, PCIT)
ffffffff80e4a567 : Return(CRES)
```

Using Breakpoints

In the following example, you will break into the AMLI Debugger before the method _BST is executed.

Even if you have located a _BST object, you should verify that it is indeed a method. You can use the **!amli dns** extension to do this.

```
kd> !amli dns /s \_sb.pci0.isa.bat1._bst

ACPI Name Space: \_SB.PCI0.ISA.BAT1._BST (c29c2044)
Method(_BST:Flags=0x0,CodeBuff=c29c20a5,Len=103)
```

Now you can use the **!amli bp** command to place the breakpoint:

```
kd> !amli bp \_sb.pci0.isa.bat1._bst
```

You may also want to place breakpoints within the method. You could use the **!amli u** command to disassemble _BST and then place a breakpoint on one of its steps:

```
kd> !amli u _sb.pci0.isa.bat1._bst

ffffffffc29c20a5: Acquire(\_SB_.PCI0.ISA_.EC0_.MUT1, 0xffff)
ffffffffc29c20c0: Store("CMBatt - _BST.BAT1", Debug)
ffffffffc29c20d7: \_SB_.PCI0.ISA_.EC0_.CPOL()
ffffffffc29c20ee: Release(\_SB_.PCI0.ISA_.EC0_.MUT1)
ffffffffc29c2107: Return(PBST)

kd> !amli bp c29c20ee
```

Responding to a Triggered Breakpoint

In the following example, the method _WAK is running and then encounters a breakpoint:

```
Running \_WAK method
Hit Breakpoint 0.
```

Use the **!amli ln** extension to see the nearest method to the current program counter. The following example is showing addresses in 32-bit form:

```
kd> !amli ln
c29accf5: \_WAK
```

The **!amli lc** extension displays all the active contexts:

```
kd> !amli lc
 Ctxt=c18b6000, ThID=00000000, Flgs=A-QC-W----, pbOp=c29bf8fe, Obj=\_SB.PCI0.ISA.EC0._Q09
*Ctxt=c18b4000, ThID=c15a6618, Flgs=----R-----, pbOp=c29accf5, Obj=\_WAK
```

This shows that the active contexts are associated with the methods _Q09 and _WAK. The current context is _WAK.

Now you can use the **!amli r** command to display more details about the current context. From this you can see useful thread and stack information, as well as arguments passed to _WAK and the local data objects.

```
kd> !amli r
Context=c18b4000*, Queue=00000000, ResList=00000000
ThreadID=c15a6618, Flags=00000010
StackTop=c18b5eec, UsedStackSize=276 bytes, FreeStackSize=7636 bytes
LocalHeap=c18b40c0, CurrentHeap=c18b40c0, UsedHeapSize=88 bytes
Object=\_WAK, Scope=\_WAK, ObjectOwner=c18b4108, SyncLevel=0
AsyncCallBack=ff06b5d0, CallBackData=0, CallBackContext=c99efddc

MethodObject=\_WAK
c18b40e4: Arg0=Integer(:Value=0x00000001[1])
c18b5f3c: Local0=Unknown()
c18b5f54: Local1=Unknown()
c18b5f6c: Local2=Unknown()
c18b5f84: Local3=Unknown()
c18b5f9c: Local4=Unknown()
c18b5fb4: Local5=Unknown()
c18b5fcc: Local6=Unknown()
c18b5fe4: Local7=Unknown()
c18b4040: RetObj=Unknown()
```

Tracing, Stepping, and Running AML Code

If you want to trace through the code, you can turn on full tracing information by using the **!amli set** extension as follows:

```
kd> !amli set spewon verboseon traceon
```

Now you can step through the AML code, watching the code execute line by line. The **p** command steps over any function calls. The **t** command will step into function calls.

```
AMLI(? for help)-> p

c29bfcb7: Store(\_SB_.PCI0.ISA_.ACAD.CHAC(SEL0=0x10e1)
c29c17b1: {
c29c17b1: | Store(LGreater(And(Arg0=0x10e1,0xf0,)=0xe0,0x80)=0xffffffff,Local0)=0xffffffff

AMLI(? for help)-> p

c29c17bb: | If(LNot(LEqual(Local0=0xffffffff,ACP_=0xffffffff)=0xffffffff)=0x0)
c29c17ce: | {
c29c17ce: | | Return(Zero)
c29c17d0: | }
c29c17d0: },Local1)=0x0

AMLI(? for help)-> t

c29bfcd4: Store(\_SB_.PCI0.ISA_.BAT1.CHBP(SEL0=0x10e1)
c29c293d: {
c29c293d: | Store("CMBatt - CHBP.BAT1",Debug)String(:Str="CMBatt - CHBP.BAT1")="CMBatt - CHBP.BAT1"
```

You may also run methods from within the AMLI Debugger if you choose. For example, you might evaluate the status of the LNKA device by running its control method _STA:

```
AMLI(? for help)-> run \_sb.lnka._sta
PCI OpRegion Access on region c29b2268 device c29b2120

\_SB.LNKA._STA completed successfully with object data:
Integer(:Value=0x0000000b[11])
```

See Also

The AMLI Debugger

Other ACPI Debugging Extensions

12/5/2018 • 2 minutes to read • Edit Online

The following extension commands are useful for debugging problems with an Advanced Configuration and Power Interface (ACPI) BIOS:

- **!acpicache** displays all of the ACPI tables cached by the hardware application layer (HAL)
- **!acpiinf** displays information on the configuration of the ACPI
- **!acpiirqarb** displays the contents of the ACPI IRQ arbiter structure
- **!facs** displays a Firmware ACPI Control Structure
- **!fadt** displays a Fixed ACPI Description Table
- **!mapic** displays an ACPI Multiple APIC Table
- **!nsobj** displays an ACPI namespace object
- **!nstree** displays a section of the ACPI namespace tree
- **!rsdt** displays the ACPI Root System Description Table

For a complete list of ACPI-related extensions, see **!acpikd.help**.

For details on the **!amli***xxx* extensions, see The AMLI Debugger.

NDIS Debugging

12/5/2018 • 2 minutes to read • Edit Online

This section includes:

Overview of NDIS Debugging

Preparing for NDIS Debugging

Enabling NDIS Debug Tracing

Overview of NDIS Debugging

1/3/2019 • 2 minutes to read • Edit Online

The two primary tools for debugging a network driver are debug tracing and the Network Driver Interface Specification (NDIS) extensions. For more information on debug tracing, see Enabling NDIS Debug Tracing. For more information on the NDIS debugging extensions, see NDIS Extensions, which provides a complete list of the extension commands found in the extension module Ndiskd.dll. The extension module appears in the winxp directory.

An additional tool for debugging a network driver is the collection of regular debugging extensions, which are useful for obtaining debugging information. For example, entering **!stacks 2 ndis!** displays all threads in the stack beginning with **ndis!**. This information can be useful for debugging hangs and stalls.

There is also an NDIS-specific bug check code, bug check 0x7C (BUGCODE_NDIS_DRIVER). For a complete list of its parameters, see **Bug Check 0x7C**.

Another useful tool for testing an NDIS driver is NDIS Verifier. For more information, consult the NDIS Verifier topic in the Windows Driver Kit (WDK) documentation.

Preparing for NDIS Debugging

12/5/2018 • 2 minutes to read • Edit Online

To debug NDIS components, a checked version of Ndis.sys is required on the target computer. However, instead of installing an entire checked-build operating system, you can copy the checked version of Ndis.sys onto an otherwise free-build operating system. Before you copy the checked version of Ndis.sys to the target computer, you must disable Windows File Protection (WFP). To ensure that WFP is disabled, start the operating system in safe mode.

The NDIS symbols are publicly available on the Microsoft symbol store. For details on how to access this, see Microsoft Public Symbols.

Enabling NDIS Debug Tracing

12/5/2018 • 2 minutes to read • Edit Online

NDIS debug tracing is the primary method for debugging NDIS drivers. When you set up NDIS debug tracing, you are actually enabling one or more levels of DbgPrint statements with NDIS. The resulting information is sufficient for debugging most network driver problems.

You can enable debug tracing by setting appropriate registry values. For details, see Enabling NDIS Debug Tracing By Setting Registry Values.

Setting registry values to enable debug tracing works even if the host computer does not have the checked version of the Ndis.sys symbols installed.

Enabling NDIS Debug Tracing By Setting Registry Values

12/5/2018 • 2 minutes to read • Edit Online

You can enable different levels of debug tracing in various NDIS components by editing the registry. Typically, you should add the following entries and values to the **HKLM\SYSTEM\CurrentControlSet\Services\NDIS\Parameters** registry key:

```
"DebugLevel"=dword:00000000
"DebugSystems"=dword:000030F3
"DebugBreakPoint"=dword:00000001
```

The following values are acceptable for **DebugBreakPoint**, **DebugLevel** and **DebugSystems**:

DebugBreakPoint
Controls whether an NDIS driver will automatically break into the debugger. If this value is set to 1, NDIS will break into the debugger when a driver enters Ndis.sys's **DriverEntry** function.

DebugLevel
Selects the level or amount of debug tracing in the NDIS components that you select with the **DebugSystems** value. The following values specify levels that you can select:

LEVEL	DESCRIPTION	VALUE
DBG_LEVEL_INFO	All available debug information. This is the highest level of trace.	0x00000000
DBG_LEVEL_LOG	Log information.	0x00000800
DBG_LEVEL_WARN	Warnings.	0x00001000
DBG_LEVEL_ERR	Errors.	0x00002000
DBG_LEVEL_FATAL	Fatal errors, which can cause the operating system to crash. This is the lowest level of trace.	0x00003000

DebugSystems
Enables debug tracing for specified NDIS components. This corresponds to using the **!ndiskd.dbgsystems** extension. The following values specify the NDIS components that you can select:

COMPONENT	DESCRIPTION	VALUE
DBG_COMP_INIT	Handles adapter initialization.	0x00000001

COMPONENT	DESCRIPTION	VALUE
DBG_COMP_CONFIG	Handles adapter configuration.	0x00000002
DBG_COMP_SEND	Handles sending data over the network.	0x00000004
DBG_COMP_RECV	Handles receiving data from the network.	0x00000008
DBG_COMP_PROTOCOL	Handles protocol operations.	0x00000010
DBG_COMP_BIND	Handles binding operations.	0x00000020
DBG_COMP_BUSINFO	Handles bus queries.	0x00000040
DBG_COMP_REG	Handles registry operations.	0x00000080
DBG_COMP_MEMORY	Handles memory management.	0x00000100
DBG_COMP_FILTER	Handles filter operations.	0x00000200
DBG_COMP_REQUEST	Handles requests.	0x00000400
DBG_COMP_WORK_ITEM	Handles work-item operations.	0x00000800
DBG_COMP_PNP	Handles Plug and Play operations.	0x00001000
DBG_COMP_PM	Handles power management operations.	0x00002000
DBG_COMP_OPENREF	Handles operations that open reference objects.	0x00004000
DBG_COMP_LOCKS	Handles locking operations.	0x00008000
DBG_COMP_RESET	Handles resetting operations.	0x00010000
DBG_COMP_WMI	Handles Windows Management Instrumentation operations.	0x00020000

COMPONENT	DESCRIPTION	VALUE
DBG_COMP_CO	Handles Connection-Oriented NDIS.	0x00040000
DBG_COMP_REF	Handles reference operations.	0x00080000
DBG_COMP_ALL	Handles all NDIS components.	0xFFFFFFFF

You can select more than one NDIS component. If more than one component is selected, combine the data values with an OR operator. For example, to select DBG_COMP_PNP, DBG_COMP_PM, DBG_COMP_INIT and DBG_COMP_CONFIG, you would combine the corresponding values (0x1000, 0x2000, 0x1, and 0x2) to obtain the value 0x3003, and then set it in the registry thus:

```
"DebugSystems"=dword:00003003
```

Whenever you change registry values for debug tracing, you must restart your computer for the new settings to take effect.

Kernel Streaming Debugging

12/5/2018 • 2 minutes to read • Edit Online

This section includes:

Overview of Kernel Streaming Debugging

Analyzing a Video Stream Stall

Analyzing a Capture Stall

Live Local Debugging

Graph Analysis with Unloadable Modules

Using !ks.graph

Overview of Kernel Streaming Debugging

12/5/2018 • 2 minutes to read • Edit Online

Kernel streaming debugging extensions can be found in the extension module Ks.dll.

You can use Ks.dll to help debug KS, AVStream, port class and stream class drivers.

For a complete list of the extension commands in Ks.dll, see Kernel Streaming Extensions (Ks.dll).

If you wish to use these extensions with Windows 2000, you must copy Ks.dll from the kdexts directory to the w2kfre directory.

Analyzing a Video Stream Stall

12/5/2018 • 2 minutes to read • Edit Online

This section shows how to analyze a video stream stall. All sample output is generated from an AVStream minidriver after an independent hardware vendor (IHV) driver bug has been introduced.

Determining the Cause

Debugging a Processing Stall

Using Logging to Track Important Events

Interpreting Bug Check 0xCB

Determining the Cause of a Video Stream Stall

12/5/2018 • 2 minutes to read • Edit Online

There are two basic causes for a video stream stall:

- **A hang.** Either a user-mode thread or a kernel-mode thread is not being released by the driver.
- **A stall.** This is the result of a problem with a component in the streaming path. Some possibilities include:
  - The capture driver is not completing packets. In this case, either a driver component or the hardware might be the source of the stall.
  - The capture driver has no packets to complete. In this case, the buffers might be stalled in a codec or other downstream component.

If you can reproduce the problem, attach a debugger at this point to determine which is the actual cause.

To determine if the problem is a hang

1. Attach a user-mode debugger to the application and look for blocked user-mode threads.
2. Determine whether the application is responsive. Can the graph be paused? Can the graph be stopped? Does streaming restart if the graph is stopped and restarted?
3. If the application is non-responsive, attempt to end the task by using Task Manager. If this fails, there is a kernel-mode hang.

To determine if the problem is a stall

1. Determine where the samples are in the graph. This can be done locally or in a kernel-mode debugging session.
2. Determine whether samples are flowing downstream. If you can reproduce the bug in GraphEdit, place an intermediate filter in the graph to display samples.
3. Determine if the processing routine is being called. This can be done by attaching a kernel-mode debugger and setting a breakpoint in this routine.

Debugging a Processing Stall

12/5/2018 • 2 minutes to read • Edit Online

Begin by finding the relevant pin. In a hypothetical case, the relevant video capture pin has address **8160DDE0**, so we use the **!ks.dump** extension command on this address to get more details:

```
kd> !ks.dump 8160DDE0 7
Pin object 8160DDE0 [CKsPin = 8160DD50]
    DeviceState    KSSTATE_RUN
    ClientState    KSSTATE_RUN
    CKsPin object 8160DD50 [KSPIN = 8160DDE0]
        State                   KSSTATE_RUN
        Processing Mutex        8160DFD0 is not held
        And Gate &              8160DF88
        And Gate Count          1
```

First, determine if the pin is in the appropriate state and whether the processing mutex is being held by another thread. In this case, the pin state is **KSSTATE_RUN**, as it should be, and the processing mutex is not being held, so we next use the **!ks.dumpqueue** extension to determine if there are frames available:

```
kd> !ks.dumpqueue 8160DDE0 7
Queue 8172D5D8:
 Frames Received  : 763
    Frames Waiting   : 5
...<this part of display not shown>...
Queue 8172D5D8:
 Frame Header 81B77E60:
        Irp = 816EE008
        Refcount = 1
    Frame Header 81A568D0:
        Irp = 816DE008
        Refcount = 0
    Frame Header 81844ED8:
        Irp = FFA0F650
        Refcount = 0
    Frame Header 8174B0B0:
        Irp = FFABB460
        Refcount = 0
    Leading Edge:
        Stream Pointer 8183EA58 [Public 8183EA90]:
            Frame Header = 81B77E60
...<this part of display not shown>...
```

In the above partial display of the **!ks.dumpqueue** output, we see that there are five frames waiting, or available. Are these frames ahead of or behind the leading edge? In the **!ks.dumpqueue** display, the frames are always listed from oldest to newest. The frame header of the leading edge matches that of the first frame listed, the oldest frame. Thus all of the available frames are ahead of the leading edge.

If this were not the case, and instead all of the frames were behind the leading edge, and they had a reference count due to clone pointers, the problems most likely originate with either the hardware or the driver's programming of hardware. Make sure that the hardware is signaling buffer completions (check interrupts and DPCs) and determine that the driver is responding appropriately to those notifications (by deleting clones upon buffer completion, for example).

If, as in our example, all of the frames are ahead of the leading edge, the problem is almost certainly a software

issue. Further information can be obtained by looking at the pin's And gate.

Interpreting the And Gate

The pin's And gate controls processing. If the gate count is one, processing can occur. Obtain the current status of the And gate by using the **!ks.dump** extension:

```
kd> !ks.dump 8160DDE0 7
Pin object 8160DDE0 [CKsPin = 8160DD50]
    DeviceState    KSSTATE_RUN
    ClientState    KSSTATE_RUN
    CKsPin object 8160DD50 [KSPIN = 8160DDE0]
        State                   KSSTATE_RUN
        Processing Mutex        8160DFD0 is not held
        And Gate &              8160DF88
        And Gate Count          1
```

Because the gate count is one, the And gate is open. In this case, investigate the following potential causes for the processing stall:

- The process dispatch incorrectly returned STATUS_PENDING.
- The data availability case is missing a KsPinAttemptProcessing call.

Using Logging to Track Important Events

7/13/2019 • 3 minutes to read • Edit Online

In general, data is moved downstream only by triggering events, the minidriver's processing, and buffer completions. To isolate the cause of a hang or stall:

- Check for mismatched **KsGate*Xxx*** calls.
- Check for omitted **Ks*Xxx*AttemptProcessing** calls.
- Look for problems in code related to triggering events, including code that either references the pin flags for the problem stream or that calls **KsPinAttemptProcessing**.
- Look for problems in the code related to the processing dispatch, in particular where it queues to hardware and where clone pointers are created.
- Look for problems in the code related to the driver's deferred procedure call (DPC), especially where buffers are completed or any calls are made to KsStreamPointerDelete.
- Look for problems in the startup code for the stream.

The most effective way to collect this information is by logging everything in the affected region, including processing, buffer acquisition (such as cloning and programming hardware), buffer release (such as deleting clones), and any gate manipulations. Most of this information is highly timing dependent and requires memory-based logging or ETW.

To maintain a rolling memory-based log, use the following code:

```
typedef struct _LOGENTRY {
    ULONG Tag;
    ULONG Arg[3];
} LOGENTRY, *PLOGENTRY;
#define LOGSIZE 2048
LONG g_LogCount;
LOGENTRY g_Log [LOGSIZE];
#define LOG(tag,arg1,arg2,arg3) do { \
    LONG i = InterlockedIncrement (&g_LogCount) % LOGSIZE; \
    g_Log [i].Tag = tag; \
    g_Log [i].Arg [0] = (ULONG)(arg1); \
    g_Log [i].Arg [1] = (ULONG)(arg2); \
    g_Log [i].Arg [2] = (ULONG)(arg3); \
} while (0)
```

Then, use a simple "dc g_Log" to view the contents of the **g_Log** array in the debugger.

The following example uses the above memory-based scheme to determine the cause of a processing stall. Output is from an AVStream streaming scenario in graphedt. The following minidriver events were logged:

ABBREVIATION	DESCRIPTION
Strt	This event occurs when the minidriver first queues buffers for the device from within the minidriver's *Start* dispatch.

ABBREVIATION	DESCRIPTION
Prc<	This event occurs at the start of the minidriver's *Process* dispatch.
AddB	This event occurs when the minidriver queues buffers to the device from within its *Process* dispatch.
DPC<	This event occurs at the start of the minidriver's *CallOnDPC*. It indicates buffer completion.
Atmp	This event occurs when the minidriver calls from within the DPC to **KsPinAttemptProcessing**.
Dele	This event occurs when the minidriver calls from within the DPC to delete a clone stream pointer.

Log excerpts are as follows:

```
f9494b80  3c435044 816e2c90 00000000 00000000  DPC<.,n.........
f9494b90  656c6544 816e2c90 81750260 00000000  Dele.,n.`.u.....
f9494ba0  706d7441 816e2c90 ffa4d418 00000000  Atmp.,n.........
f9494bb0  3c637250 819c1f00 00000000 00000000  Prc<............
f9494bc0  42646441 819c1f00 ffa2eb08 00000000  AddB............
f9494bd0  3c435044 816e2c90 00000000 00000000  DPC<.,n.........
f9494be0  656c6544 816e2c90 ffa80348 00000000  Dele.,n.H.......
f9494bf0  706d7441 816e2c90 ffa4d418 00000000  Atmp.,n.........
f9494c00  3c637250 819c1f00 00000000 00000000  Prc<............
f9494c10  42646441 819c1f00 ffa3d9b8 00000000  AddB............
```

This first log excerpt is representative of the normal streaming state. In the first line, the minidriver's *CallOnDPC* is called to complete a buffer (*DPC<*). The buffer is deleted (*Dele*), and **KsPinAttemptProcessing** is called to move the leading edge forward, if there are any unprocessed buffers in the queue (*Atmp*). In this case, there were, as can be seen by the call to the process dispatch (*Prc<*). More buffers are added to the queue (*AddB*), and the whole scenario repeats.

This next excerpt includes the last entries in the log right before the stall occurred.

```
f949b430  3c435044 816e2c90 00000000 00000000  DPC<.,n.........
f949b440  656c6544 816e2c90 ffac4de8 00000000  Dele.,n..M......
f949b450  706d7441 816e2c90 ffa4d418 00000000  Atmp.,n.........
f949b460  3c435044 816e2c90 00000000 00000000  DPC<.,n.........
f949b470  656c6544 816e2c90 816ffc80 00000000  Dele.,n...o.....
f949b480  706d7441 816e2c90 ffa4d418 00000000  Atmp.,n.........
f949b490  3c435044 816e2c90 00000000 00000000  DPC<.,n.........
f949b4a0  656c6544 816e2c90 ffa80348 00000000  Dele.,n.H.......
f949b4b0  706d7441 816e2c90 ffa4d418 00000000  Atmp.,n.........
f949b4c0  3c435044 816e2c90 00000000 00000000  DPC<.,n.........
f949b4d0  656c6544 816e2c90 8174e1c0 00000000  Dele.,n...t.....
f949b4e0  706d7441 816e2c90 ffa4d418 00000000  Atmp.,n.........
```

In this example, several buffers are being completed (indicated by the repeated instances of *DPC<*), but there are no unprocessed buffers in the queue, so the process dispatch is not being called (indicated by the absence of *Prc<*). In fact, all of the processed buffers in the queue have been completed, apparently before any new unprocessed

buffers could be added. Because the application is already running (so that *Start* will not be called) and no calls are being made to *CallOnDPC* (because there are no processed buffers ready to be completed), any new buffers are apparently accumulating ahead of the leading edge, waiting to be processed, with nothing initiating processing.

The problem is that the KSPIN_FLAG_DO_NOT_INITIATE_PROCESSING flag has been set. When this flag is set, processing occurs only through a call to *Start* or *CallOnDPC*. If this flag is not set, processing will be initiated whenever new buffers are added to the queue.

Interpreting Bug Check 0xCB

12/5/2018 • 2 minutes to read • Edit Online

The most common bug check code associated with debugging a video stream stall is Bug Check 0xCB (DRIVER_LEFT_LOCKED_PAGES_IN_PROCESS). For a detailed list of its parameters, see **Bug Check 0xCB**.

The message displayed when the bug check occurs will point to Ks.sys as the cause.

```
Use !analyze -v to get detailed debugging information.
BugCheck CB, {f90c6ae0, f9949215, 81861788, 26}
Probably caused by : ks.sys ( ks!KsProbeStreamIrp+333 )
```

As suggested, use **!analyze -v** to get more detailed information.

```
kd> !analyze -v
DRIVER_LEFT_LOCKED_PAGES_IN_PROCESS (cb)
Caused by a driver not cleaning up completely after an I/O.
When possible, the guilty driver's name (Unicode string) is printed on
the bugcheck screen and saved in KiBugCheckDriver.
Arguments:
Arg3: 81861788, A pointer to the MDL containing the locked pages.
```

Now, use the **!search** extension to find the virtual addresses that are associated with the MDL pointer.

```
kd> !search 81861788
Searching PFNs in range 00000001 - 0000FF76 for [FFFFFFFF81861788 - FFFFFFFF81861788]

Pfn      Offset   Hit      Va       Pte
- - - - - - - - - - - - - - - - - - - - - - - - - - - -
000008A7 00000B0C 81861788 808A7B0C C020229C
00000A04 00000224 16000001 80A04224 C0202810
...
00001732 000009B4 81861788 817329B4 C0205CC8
```

For each virtual address (VA) found, look for an IRP signature. Do this by using the **dd** command with the VA minus one DWORD.

```
kd> dd 808A7B0C-4 l4
808a7b08  f9949215 81861788 00000026 00000000
kd> $ Not an Irp
kd> dd 80A04224-4 l4
80a04220  00000000 00000000 00000000 00000000
kd> $ Not an Irp
kd> dd 817329B4-4 l4
817329b0  01900006 81861788 00000070 ffa59220
kd> $ Matches signature
```

After a VA with an IRP signature has been found, use the **!irp** extension to find out what driver is pending on this IRP.

```
kd> !irp 817329b0 7
Irp is active with 2 stacks 2 is current (= 0x81732a44)
 Mdl = 81861788 System buffer = ffa59220 Thread 00000000:  Irp stack trace.
>[  e, 0]   1  1 81a883c8 81ae6158 00000000-00000000    pending
 \Driver\TESTCAP
                       Args: 00000070 00000000 002f4017 00000000
```

In this case, \Driver\TESTCAP is the likely cause of the bug check.

Analyzing a Capture Stall

12/5/2018 • 2 minutes to read • Edit Online

The following is an artificially created scenario that simulates a capture stall. This is a particularly valuable scenario since similar situations frequently occur in stress testing. The scenario is as follows:

The Windows recording component Sndrec32 is recording from the primary capture device, in this case a Creative SBLive wave device. For a period of time, it records normally; however, the graph stalls at 8.50 seconds because we have explicitly caused portcls not to complete capture IRPs for purposes of this test.

The application shows running, but the stream position is not advancing. Position is halted at 8.50 seconds.

Since the primary capture device on this machine is a PCI sound card, first use the **!ks.pciaudio** command to try and determine a starting point. Use a flag value of 1 to request a display of all running streams:

```
kd> !pciaudio 1

1 Audio FDOs found:
 Functional Device 8121c030 [\Driver\emu10k]
        Wave Cyclic Streams:
            Pin 812567c0 RUN [emu10k1m!CMiniportWaveCyclicStreamSBLive ff9ec7f8]
```

In this case, there is only one PCI audio device and it is serviced by the Intel emu10k driver (\Driver\emu10k). This driver currently has a single running stream (0x812567C0). Now you can use **!ks.graph** to view the kernel graph. Set *Level* and *Flags* both to 7 to obtain maximum detail on the stall:

```
kd> !graph 812567c0 7 7
Attempting a graph build on 812567c0...  Please be patient...
Graph With Starting Point 812567c0:
"emu10k" Filter ff9ebb98, Child Factories 5
    Output Factory 0:
        Pin 812567c0 (File 811c6630, -> "splitter" 811df960) Irps(q/p) = 8, 0
 Queued: 81255418 811df008 81252008 81255280 81250b30 ffa1fe70 81252e70 ffa01d98
```

The above shows the details for factory 0. The emu10k output pin 0x812567C0 is connected to the splitter input pin 0x811DF960. There are eight IRPs queued to emu10k's output pin. The output from **!ks.graph** continues as follows:

```
"splitter" Filter ffb18890, Child Factories 2
    Output Factory 0:
        Pin 811df430 (File ffa55f90) Irps(q/p) = 10, 0
 Queued: ffadd008 ffa73b00 ffa1e998 811de310 ffa54370 ffaaf008 811dee70 81250e70 811de580 811de8c0
```

There are ten IRPs queued to splitter's output pin.

```
    Input Factory 1:
        Pin 811df960 (File 81187820, <- "emu10k" 812567c0) Irps(q/p) = 0, 8
            Pending: 81255418 811df008 81252008 81255280 81250b30 ffa1fe70 81252e70 ffa01d98
```

Splitter's input pin has no queued IRPs; however, it is waiting for the eight from emu10k to enter the queue.

```
Analyzing a Hung Graph From 812567c0:

Suspect Filters (For a Hung Graph):
 "emu10k" Filter ff9ebb98 or class "PortCls Wave Cyclic" is suspect.
        Reasons For This Analysis:
            - No critical pin has less than 8 queued Irps
            - Downstream "splitter" pin 811df960 is starved
        Irps to check:
 81255418 811df008 81252008 81255280 81250b30 ffa1fe70 81252e70 ffa01d98
```

From this information, the analyzer suggests that either emu10k or WaveCyclic may be at fault. It also provides a list of the suspect IRPs; these are the IRPs that are queued to emu10k's input pin. If any of those IRPs were to complete, splitter would copy data and complete an IRP and the graph would progress. For some reason, emu10k is not completing those capture Irps.

Live Local Debugging

12/5/2018 • 2 minutes to read • Edit Online

In Microsoft Windows XP and later operating systems, it is possible to do local kernel debugging by starting the kernel debugger (KD) or WinDbg with the **-kl** command line option:

```
kd [-y SymbolPath] -kl
```

or

```
windbg [-y SymbolPath] -kl
```

In Windows Vista and later, local kernel debugging requires the computer to be booted with the **/debug** option. Open a Command Prompt Window as Administrator, and enter **bcdedit /debug on**. Reboot the computer.

> **IMPORTANT**
> Before using BCDEdit to change boot information you may need to temporarily suspend Windows security features such as BitLocker and Secure Boot on the test PC. Re-enable these security features when testing is complete and appropriately manage the test PC, when the security features are disabled.

In Windows Vista and later, local kernel debugging requires the debugger to be run as Administrator.

Live local debugging is extremely useful for debugging issues that are difficult to reproduce when the debugger is attached; however, anything that requires knowledge of time sensitive information, including packet, IRP, and SRB data, is unlikely to work unless the problem is a hang or a stall.

When performing local debugging, consider the following variables:

- **Overall states.** For example, is the stream running? Is the stream paused?
- **Packet counts.** For example, are there IRPs queued to the stream?
- **Changes in packet counts.** Is the stream moving?
- **Changes in packet lists.**
- **Hung kernel threads.**

Consider the last of these.

Examining a Hung Thread in LKD

First, use the **!process 0 0** extension to identify the process containing the hung thread. Then, issue **!process** again for more information about that thread:

```
lkd> !process 816a550 7
        THREAD 81705da8  Cid 0b5c.0b60  Teb: 7ffde000 Win32Thread: e1b2d890 WAIT: (Suspended)
        IRP List:
            816c9ad8: (0006,0190) Flags: 00000030  Mdl: 00000000
        Start Address kernel32!BaseProcessStartThunk (0x77e5c650)
        Win32 Start Address 0x0101c9be
        Stack Init f50bf000 Current f50bea74 Base f50bf000 Limit f50b9000 Call 0
        Priority 10 BasePriority 8 PriorityDecrement 0
```

The threads are not displayed, but the stack addresses are. Using the **dds** (or **ddq**) command on the current address on the stack yields a starting point for further investigation, because it specifies which process is calling.

```
lkd> dds f50bea74
f50bea74  f50bebc4
f50bea78  00000000
f50bea7c  80805795 nt!KiSwapContext+0x25
f50beab4  8080ece0 nt!KeWaitForSingleObject
f50beabc  f942afda ks!CKsQueue::CancelAllIrps+0x14
f50bead8  f94406c4 ks!CKsQueue::SetDeviceState+0x170
f50beb00  f943f6f1 ks!CKsPipeSection::DistributeDeviceStateChange+0x1d
f50beb24  f943fb1e ks!CKsPipeSection::SetDeviceState+0xb2
```

Graph Analysis with Unloadable Modules

12/5/2018 • 3 minutes to read • Edit Online

This section describes a scenario that may affect you if you are working with unloadable modules such as KMixer.

After loading, the extension module initializes at the first command usage. At initialization, the extension module checks whether every module is loaded and has correct symbols. If any individual module is unloaded or has incorrect symbols loaded, the extension disables the library extension which handles identification, dumping, etc. for that module. In this case, you need to manually re-enable the disabled module.

The above situation may occur if you load the extension at boot time. Specifically, you may encounter this scenario if you load Ks.dll and then issue a **.reboot** command. Or, it could happen if you break into the debugger during boot and load Ks.dll at that point.

In the following example, we are capturing two streams (sndrec32) from a Telex USB microphone. Breaking on **splitter!FilterProcess** and running **!ks.graph** on splitter's filter yields:

```
kd> !graph ffa0c6d4 7
Attempting a graph build on ffa0c6d4...  Please be patient...
Graph With Starting Point ffa0c6d4:
"usbaudio" Filter ffaaa768, Child Factories 2
    Output Factory 0:
        Pin ffb1caf0 (File 811deeb8, -> "splitter" ffa8b008) Irps(q/p) = 7, 1
"splitter" Filter ffa0c660, Child Factories 2
    Output Factory 0:
        Pin 81250008 (File ffb10028) Irps(q/p) = 3, 0
        Pin 811df9c0 (File ffaaf2f0) Irps(q/p) = 3, 0
    Input Factory 1:
        Pin ffa8b008 (File ffb26d68, <- "usbaudio" ffb1caf0) Irps(q/p) = 1, 7
```

In this example, KMixer has been loaded and connected to splitter, but Kmixer does not appear in the graph. There are IRPs queued to splitter's output pin, yet the **!ks.graph** command is unable to backtrace and discover KMixer. Issue a **!ks.libexts details** command to investigate further:

```
kd> !libexts details
## LibExt Details:
------------------------------------------------
LibExt "portcls!" :
    Status :  ACTIVE
    This is the port class library extension to the KS DLL.  It supports
    dumping wave cyclic, wave pci, irp streams, and several other upper
    level structures.
    Commands Exported: pciaudio
    Help : pchelp
    Hooks : dump dumpqueue dumpcircuit conv(file) conv(device) graph
LibExt "STREAM!" :
    Status :  ACTIVE
    This is the stream class library extension to the KS DLL.  It supports
    dumping device extensions, filters, streams, and SRBs.
    Hooks : dump enumdevobj graph
LibExt "kmixer!" :
    Status :  INACTIVE
    This is the KMIXER extension to the KS DLL.  It supports
    virtually nothing at this point!
    Hooks : dump graph
```

According to the above output, the KMixer section of the extension is currently disabled (Status : INACTIVE). Since the extension module was first used in a context in which KMixer was not loaded, Ks.dll has disabled the KMixer section of the extension to prevent time-consuming references to an unloaded module.

To enable KMixer explicitly, you can use **!ks.libexts** with the **enable** parameter, as follows:

```
kd> !libexts enable kmixer
LibExt "kmixer" has been enabled.

kd> !graph ffa0c6d4 7
Attempting a graph build on ffa0c6d4...  Please be patient...
Graph With Starting Point ffa0c6d4:
"usbaudio" Filter ffaaa768, Child Factories 2
    Output Factory 0:
        Pin ffb1caf0 (File 811deeb8, -> "splitter" ffa8b008) Irps(q/p) = 7, 1
"splitter" Filter ffa0c660, Child Factories 2
    Output Factory 0:
        Pin 81250008 (File ffb10028, -> "kmixer" 8123c000) Irps(q/p) = 3, 0
        Pin 811df9c0 (File ffaaf2f0, -> "kmixer" 81236000) Irps(q/p) = 3, 0
    Input Factory 1:
        Pin ffa8b008 (File ffb26d68, <- "usbaudio" ffb1caf0) Irps(q/p) = 1, 7
"kmixer" Filter ffa65b70, Child Factories 4
    Input Factory 2:
        Pin 81236000 (File ffaaf7d0, <- "splitter" 811df9c0) Irps(q/p) = 0, 0
    Output Factory 3:
        Pin 81252d00 (File 811df1d8) Irps(q/p) = 10, 0
"kmixer" Filter ffb03808, Child Factories 4
    Input Factory 2:
        Pin 8123c000 (File ffb10130, <- "splitter" 81250008) Irps(q/p) = 0, 0
    Output Factory 3:
        Pin ffa1e9c0 (File 81253468) Irps(q/p) = 10, 0
```

KMixer now appears as expected in the capture graph.

Using !ks.graph

12/5/2018 • 3 minutes to read • Edit Online

The **!ks.graph** command is one of most powerful extension commands in the kernel streaming extension module. This command displays a picture of an entire graph in kernel mode from any given starting point.

Before running **!ks.graph**, you may want to enable all library extensions that are capable of being active. To do this, issue a **!ks.libexts enableall** command. The output of **!ks.graph** will be a textual description of the kernel mode graph in topologically sorted order. Here is an example:

```
kd> !graph ffa0c6d4 7
Attempting a graph build on ffa0c6d4...  Please be patient...
Graph With Starting Point ffa0c6d4:
"usbaudio" Filter ffaaa768, Child Factories 2
    Output Factory 0:
        Pin ffb1caf0 (File 811deeb8, -> "splitter" ffa8b008) Irps(q/p) = 7, 1
```

This example displays a capture graph in which two Sndrec32.exe's are capturing from a Telex USB Microphone. Each individual record begins with a name (usbaudio, in the above section) and shows the filter address (0xFFAAA768) and quantity of child pin factories (2) on the filter.

Below each entry, each factory is enumerated and lists the address of each pin instance (0xFFB1CAF0), the file object (0x811DEEB8) corresponding to each instance, the direction of the connection, the destination of that connection, and the address of the destination pin (0xFFA8B008). The number of queued (7) and pending IRPs (1) for each pin is also displayed.

Connections which have forward direction symbols (->) indicate that the pin is an output pin and is connected to an input pin. Connections which have reverse direction symbols (<-), on the other hand, are input pins and show the origination of the connection. The output continues as follows:

```
"splitter" Filter ffa0c660, Child Factories 2
    Output Factory 0:
        Pin 81250008 (File ffb10028, -> "kmixer" 8123c000) Irps(q/p) = 3, 0
        Pin 811df9c0 (File ffaaf2f0, -> "kmixer" 81236000) Irps(q/p) = 3, 0
    Input Factory 1:
        Pin ffa8b008 (File ffb26d68, <- "usbaudio" ffb1caf0) Irps(q/p) = 1, 7
"kmixer" Filter ffa65b70, Child Factories 4
    Input Factory 2:
        Pin 81236000 (File ffaaf7d0, <- "splitter" 811df9c0) Irps(q/p) = 0, 0
    Output Factory 3:
        Pin 81252d00 (File 811df1d8) Irps(q/p) = 10, 0
"kmixer" Filter ffb03808, Child Factories 4
    Input Factory 2:
        Pin 8123c000 (File ffb10130, <- "splitter" 81250008) Irps(q/p) = 0, 0
    Output Factory 3:
        Pin ffa1e9c0 (File 81253468) Irps(q/p) = 10, 0
```

In order to follow the graph, use the following procedure:

To follow this graph:

1. Find the pin of interest. Consider 0xFFB1CAF0, usbaudio's output pin (factory 0).

2. Find the connected pin. In this example, this is splitter pin 0xFFA8B008.

3. Look at the connection direction and visually move that way looking for the filter name. (Remember, the list is topologically sorted.) In this example, the right-pointing arrow indicates that we need to look below this entry in the list to find the corresponding pin. The splitter filter 0xFFA0C660 is immediately below.

4. Find the destination pin address in the filter pin instance list. In this case, this address is 0xFFA8B008.

The To follow this graph: command can also be used to analyze stalled graphs from any given starting point. To do this, specify 4 in the *Flags* parameter:

```
kd> !graph 812567c0 7 4

Attempting a graph build on 812567c0...  Please be patient...

Graph With Starting Point 812567c0:

"emu10k" Filter ff9ebb98, Child Factories 5
    Output Factory 0:
        Pin 812567c0 (File 811c6630, -> "splitter" 811df960) Irps(q/p) = 8, 0
"splitter" Filter ffb18890, Child Factories 2
    Output Factory 0:
        Pin 811df430 (File ffa55f90) Irps(q/p) = 10, 0
    Input Factory 1:
        Pin 811df960 (File 81187820, <- "emu10k" 812567c0) Irps(q/p) = 0, 8

Analyzing a Hung Graph From 812567c0:

Suspect Filters (For a Hung Graph):
    "emu10k" Filter ff9ebb98 or class "PortCls Wave Cyclic" is suspect.
        Reasons For This Analysis:
            - No critical pin has less than 8 queued Irps
            - Downstream "splitter" pin 811df960 is starved
        Irps to check:
            81255418 811df008 81252008 81255280 81250b30 ffa1fe70 81252e70 ffa01d98

NOTE: The above is based on heuristic analysis.  It is not designed to be a
      replacement for an actual developer looking at this particular hang!  The
      filters listed as suspects may or may not be the actual cause of the
      stall!
```

For such output, look at the "suspects" list. These suspect filters are those that are in the critical path of progress being made in the graph. Begin debugging from that point based on the reasons that the analyzer has produced for the stall.

Note This functionality should only be used on stalled graphs! The analyzer has no way of knowing how long the graph has been in this state. Breaking into the debugger and analyzing a running graph as a stalled graph still displays a suspect filter.

SCSI Miniport Debugging

12/5/2018 • 2 minutes to read • Edit Online

This section includes:

Overview of SCSI Miniport Debugging

Extensions for Debugging SCSI Miniport Drivers

Bug Checks for SCSI Miniport Debugging

Analyzing Stalled Drivers and Time-Outs

Important Breakpoints for Analyzing Reproducible Problems

Overview of SCSI Miniport Debugging

7/13/2019 • 2 minutes to read • Edit Online

SCSI Verification

Small computer system interface (SCSI) debugging extensions can be found in two extension modules: Scsikd.dll and Minipkd.dll. For an overview of the most important extension commands in these modules, see Extensions for Debugging SCSI Miniport Drivers. For a complete list, see SCSI Miniport Extensions.

The SCSIkd.dll extension commands can be used in any version of Windows. The Minipkd.dll extension commands can only be used in Windows XP and later versions of Windows. Commands in Minipkd.dll are only applicable to miniport drivers that work with the SCSI Port driver.

To test a SCSI miniport driver, use the SCSI Verification feature of Driver Verifier. For information about Driver Verifier, see Driver Verifier in the Windows Driver Kit (WDK) documentation.

Extensions for Debugging SCSI Miniport Drivers

12/5/2018 • 4 minutes to read • Edit Online

When you debug SCSI miniport drivers, you may find the following debugger extensions useful. General debugger extensions are listed first, followed by those specific to SCSI miniport debugging.

!devobj

The **!devobj** extension displays detailed information about a DEVICE_OBJECT. If the **Current Irp** field is nonnull, this could be caused by the SCSI driver waiting for map registers.

Here is an example:

```
0: kd> !devobj 8633da70
Device object (8633da70) is for:
 adpu160m1 \Driver\adpu160m DriverObject 8633eeb8
Current Irp 860ef008 RefCount 0 Type 00000004 Flags 00000050
Dacl e129871c DevExt 8633db28 DevObjExt 8633dfd0
ExtensionFlags (0000000000)
AttachedTo (Lower) 863b2978 \Driver\PCI
Device queue is not busy.
```

!errlog

The **!errlog** extension displays the contents of any pending entries in the I/O system's error log.

!object

The **!object** extension displays information about a system object. This extension displays all SCSI devices.

For example:

```
0: kd> !object \device\scsi
Object: e12a2520  Type: (863d12c8) Directory
    ObjectHeader: e12a2508
    HandleCount: 0  PointerCount: 9
    Directory Object: e1001100  Name: Scsi

    Hash Address  Type          Name
    ---- -------  ----          ----
     04  86352040 Device        adpu160m1Port3Path0Target6Lun0
     11  86353040 Device        adpu160m1Port3Path0Target1Lun0
     13  86334a70 Device        lp6nds351
     22  862e6040 Device        adpu160m1Port3Path0Target0Lun0
     24  8633da70 Device        adpu160m1
     25  86376040 Device        adpu160m2
     34  862e5040 Device        adpu160m1Port3Path0Target2Lun0
```

!pcr

The **!pcr** extension displays detailed information about the Processor Control Region (PCR) on a processor. The information includes the items in the DPC queue, which can be useful. when you are debugging a stalled driver or a time-out.

!minipkd.help

The **!minipkd.help** extension displays a list of all of the Minipkd.dll extension commands.

If an error message similar to the following appears, it indicates that the symbol path is incorrect and does not point to the correct version of the Scsiport.sys symbols.

```
minipkd error (0) <path> ... \minipkd\minipkd.c @ line 435
```

The **.sympath (Set Symbol Path)** command can be used to display the current path and to change the path. The **.reload (Reload Module)** command will reload symbols from the current path.

!minipkd.adapter Adapter

The **!minipkd.adapter** extension displays detailed information about a specified adapter. The **Adapter** can be found by looking at the **DevExt** field in the **!minipkd.adapters** display.

!minipkd.adapters

The **!minipkd.adapters** extension displays all the adapters that work with the SCSI Port driver that have been identified by Windows, and the individual devices associated with each adapter.

Here is an example:

```
0: kd> !minipkd.adapters
Adapter \Driver\lp6nds35     DO 86334a70         DevExt 86334b28
Adapter \Driver\adpu160m     DO 8633da70         DevExt 8633db28
 LUN 862e60f8 @(0,0,0) c ev     pnp(00/ff) pow(0,0) DevObj 862e6040
 LUN 863530f8 @(0,1,0) c ev p d pnp(00/ff) pow(0,0) DevObj 86353040
 LUN 862e50f8 @(0,2,0) c ev     pnp(00/ff) pow(0,0) DevObj 862e5040
 LUN 863520f8 @(0,6,0)   ev     pnp(00/ff) pow(0,0) DevObj 86352040
Adapter \Driver\adpu160m     DO 86376040         DevExt 863760f8
```

An error message similar to the following indicates that either the symbol path is incorrect and does not point to the correct version of the Scsiport.sys symbols, or that Windows has not identified any adapters that work with the SCSI Port driver:

```
minipkd error (0) <path> ... \minipkd\minipkd.c @ line 435
```

If the **!minipkd.help** extension command returns help information successfully, the SCSI Port symbols are correct.

!minipkd.exports Adapter

The **!minipkd.exports** extension displays the addresses of the miniport exports for the specified adapter.

!minipkd.lun {LUN | Device}

The **!minipkd.lun** extension displays detailed information about a specified Logical Unit Extension (LUN). The LUN can be specified either by its address (which can be found by looking at the **LUN** field in the **!minipkd.adapters** display) or by its physical device object (which can be found in the **DevObj** field of the **!minipkd.adapters** display).

!minipkd.portconfig PortConfig

The **!minipkd.portconfig** extension displays detailed information about a specified PORT_CONFIGURATION_DATA. The **PortConfig** can be found in the **Port Config Info** field of the **!minipkd.adapter** display.

!minipkd.req {Adapter | Device}

The **!minipkd.req** extension displays information about all of the currently active requests on the specified adapter or LUN device.

!minipkd.srb SRB

The **!minipkd.srb** extension displays detailed information about a specified SCSI request block (SRB). The SRB is specified by address. The addresses of all currently active requests can be found in the **SRB** fields of the output from the **!minipkd.req** command.

!scsikd.classext [Device [Level]]

The **!scsikd.classext** extension displays detailed information about a specified class Plug and Play (PnP) device or a list of all such devices. The *Device* is the device object or device extension of the class PnP device. If *Device* is omitted, a list of all class PnP extensions is displayed.

Here is an example:

```
0: kd> !scsikd.classext

 ' !scsikd.classext 8633e3f0 '   (             ) "IBM     " / "DDYS-T09170M    " / "S93E" / "        XBY45906"
 ' !scsikd.classext 86347b48 '   (paging device) "IBM     " / "DDYS-T09170M    " / "S80D" / "        VDA60491"
  ' !scsikd.classext 86347360 '   (             ) "UNISYS  " / "003451ST34573WC " / "5786" /
"HN0220750000181300L6"
  ' !scsikd.classext 861d1898 '   (             ) "" / "MATSHITA CD-ROM CR-177" / "7T03" / ""

 usage: !classext <class fdo> <level [0-2]>
```

!scsikd.scsiext Device

The **!scsikd.scsiext** extension displays detailed information about a specified SCSI port extension. The *Device* can be the device object or device extension of either the adapter or the LUN.

Here are some examples:

```
0: kd> !scsikd.scsiext 86353040
Common Extension:
   < ..omitted.. >
Logical Unit Extension:
  Address (3, 0, 1, 0) Claimed  Enumerated Visible
  LuFlags (0x00000000):
  Retry 0x00          Key 0x008889ff
  Lock 0x00000000  Pause 0x00000000   CurrentLock: 0x00000000
  HwLuExt 0x862e6f00  Adapter 0x8633db28  Timeout 0x0000000a
  NextLun 0x00000000  ReadyLun 0x00000000
  Pending 0x00000000  Busy 0x00000000     Untagged 0x00000000
  < ..omitted.. >
Request list @0x86353200:
      Tick count is 2526
      SrbData 8615d700  Srb 8611f4fc  Irp 8611f2b8   Key 60197  <1s
      SrbData 85e72868  Srb 86100c3c Irp 861009f8   Key e29dc7  <1s

0: kd> !scsikd.scsiext 8633da70
Common Extension:
   < ..omitted.. >
Adapter Extension:
  Port 3     IsPnp VirtualSlot HasInterrupt
  LowerPdo 0x84f9fb68   HwDevExt 0x84634004   Active Requests 0x00000000
  MaxBus 0x03   MaxTarget 0x40   MaxLun 0x08
  Port Flags (0x00001000): PD_DISCONNECT_RUNNING
  NonCacheExt 0x850d4000  IoBase 0xd80f0000   Int 0x23  < ..omitted.. >
```

!scsikd.srbdata Address

The **!scsikd.srbdata** extension displays detailed information about a specified SRB_DATA tracking block.

Bug Checks for SCSI Miniport Debugging

12/5/2018 • 2 minutes to read • Edit Online

There are primarily two bug checks that arise in the course of debugging a SCSI miniport driver: bug check 0x77 (KERNEL_STACK_INPAGE_ERROR) and bug check 0x7A (KERNEL_DATA_INPAGE_ERROR). For full details of their parameters, see **Bug Check 0x77** and **Bug Check 0x7A**.

Each of these bug checks indicates that a paging error has occurred. There are three main causes for these bug checks:

- Full bus reset due to a timeout on a particular device or no activity on an adapter
- Selection time-out
- Controller errors

To determine the precise cause of the failure, begin by using the **!scsikd.classext** extension, which displays information about recently failed requests, including the SRB status, SCSI status, and sense data of the request.

```
kd> !scsikd.classext 816e96b0
Storage class device 816e96b0 with extension at 816e9768

Classpnp Internal Information at 817b4008

    Failed requests:

           Srb    Scsi
    Opcode Status Status Sense Code  Sector   Time  Stamp
    ------ ------ ------ ---------- -------- ------------
      2a     0a     02    03 0c 00  0000abcd 23:01:07.453  Retried
      28     0a     02    03 04 00  0000abcd 23:01:07.984  Retried

dt classpnp!_CLASS_PRIVATE_FDO_DATA 817b4008 -

...
```

In the previous example, opcode 0x2A indicates a write operation, and 0x28 indicates a read operation. The SCSI status in the example is 02, which indicates a check condition. The sense codes provide more error information.

As always, miniport driver developers are responsible for associating error codes from their hardware to the SRB status codes. Typically, timeouts are associated with SRB 0x0A, the code for a selection timeout. SRB 0x0e is typically associated with a full bus reset, but it can also be associated with controller errors.

Analyzing Stalled Drivers and Time-Outs

12/5/2018 • 2 minutes to read • Edit Online

When debugging a SCSI miniport driver, the three most common causes for hangs and time-outs are:

- The SCSI miniport DPC is not running
- The SCSI miniport fails to ask for the next request
- A request is not being completed by the SCSI miniport, usually because it is waiting for map registers.

If you suspect that the SCSI miniport DPC is not running, use **!pcr** to display the DPC queue for the current processor. If the SCSI port DPC routine is in the DPC queue, place a breakpoint on this routine to determine whether this routine is ever called. Otherwise, use **!scsikd.scsiext** on each device. Consider the following sample output from the **!scsikd.scsiext** extension:

```
0: kd> !scsikd.scsiext 86353040
Common Extension:
   < ..omitted.. >
Logical Unit Extension:
  Address (3, 0, 1, 0) Claimed  Enumerated Visible
  LuFlags (0x00000000):
  Retry 0x00          Key 0x008889ff
  Lock 0x00000000  Pause 0x00000000   CurrentLock: 0x00000000
  HwLuExt 0x862e6f00  Adapter 0x8633db28  Timeout 0x0000000a
  NextLun 0x00000000  ReadyLun 0x00000000
  Pending 0x00000000  Busy 0x00000000     Untagged 0x00000000

  . . .

Request list @0x86353200:
      Tick count is 2526
      SrbData 8615d700  Srb 8611f4fc  Irp 8611f2b8   Key 60197  <1s
      SrbData 85e72868  Srb 86100c3c Irp 861009f8   Key e29dc7  <1s
```

If the timeout slot is -1 and the untagged slot is nonzero, or the time-out slot is nonzero and there are requests shown, the miniport has failed to ask for the next request.

Alternatively, if the retry slot and the busy slot are nonzero, a request may not be completed by the SCSI miniport because it is waiting for map registers. Similarly, if the untagged and pending slots are nonzero, the SCSI miniport might be waiting for map registers. In either case, the address of the SCSI request block (SRB) is the address in the busy slot and the address of the request that is not being completed. For more information about the SRB, use the **!minipkd.srb** extension with this address as input.

The **!devobj** extension determines whether the SCSI miniport is waiting for map registers. Use the device object address of the device that is issuing the request as input to **!devobj**. If the current IRQ is nonzero, it is highly probable that the SCSI miniport is waiting for map registers.

Important Breakpoints for Analyzing Reproducible Problems

12/5/2018 • 2 minutes to read • Edit Online

When debugging a SCSI miniport driver, there are three routines in which it is useful to set a breakpoint:

- **scsiport!scsiportnotification**
- **scsiport!spstartiosynchronized**
- **miniport!HwStartIo**

The routine **scsiport!scsiportnotification** is called right after a request is sent to the miniport. Thus, if you set a breakpoint in **scsiport!scsiportnotification** and then run a stack backtrace using **kb 3**, you can determine whether the miniport is receiving and completing requests. If the first parameter is zero, the request has been completed. If the first parameter is nonzero, the third parameter is the address of the SCSI request block (SRB) that is not being completed, and you can use the **!minipkd.srb** extension to further analyze the situation.

Placing a breakpoint in either **scsiport!spstartiosynchronized** or **miniport!HwStartIo** will cause a break just prior to sending a request to the miniport.

Plug and Play Debugging

12/5/2018 • 2 minutes to read • Edit Online

This section includes:

Extensions for Debugging Plug and Play Drivers

Determining the Status of a Device

Device Node Status Flags

Device Manager Problem Codes

Checking for Resource Conflicts

Extensions for Debugging Plug and Play Drivers

8/27/2019 • 3 minutes to read • Edit Online

When you debug Plug and Play drivers, you may find the following debugger extensions useful.

!arbiter

Displays the current system resource arbiters. An arbiter is a piece of code that is exposed by the bus driver that arbitrates requests for resources, and attempts to solve the resource conflicts among the devices connected on that bus.

!cmreslist

Displays the CM_RESOURCE_LIST for the specified device object.

You must know the address of the CM Resource List.

Here is an example:

```
kd> !cmreslist 0xe12576e8

CmResourceList at 0xe12576e8  Version 0.0  Interface 0x1  Bus #0
  Entry 0 - Port (0x1) Device Exclusive (0x1)
    Flags (0x01) - PORT_MEMORY PORT_IO
    Range starts at 0x3f8 for 0x8 bytes
  Entry 1 - Interrupt (0x2) Shared (0x3)
    Flags (0x01) - LATCHED
    Level 0x4, Vector 0x4, Affinity 0xffffffff
```

This shows that the device with this CM resource list is using I/O Range 3F8-3FF and IRQ 4.

!dcs

This extension is obsolete -- its functionality has been subsumed by **!pci**. See the !pci 100 example later in this section.

!devext

Displays bus-specific device extension information for a variety of devices.

!devnode

Displays information about a node in the device tree.

Device node 0 (zero) is the root of the device tree.

Here is an example:

```
0: kd> !devnode 0xfffffa8003634af0
DevNode 0xfffffa8003634af0 for PDO 0xfffffa8003658590
  Parent 0xfffffa8003604010   Sibling 0xfffffa80036508e0   Child 0000000000
  InstancePath is "ROOT\SYSTEM\0000"
  ServiceName is "swenum"
  State = DeviceNodeStarted (0x308)
  Previous State = DeviceNodeEnumerateCompletion (0x30d)
  StateHistory[09] = DeviceNodeEnumerateCompletion (0x30d)
  StateHistory[08] = DeviceNodeEnumeratePending (0x30c)
  StateHistory[07] = DeviceNodeStarted (0x308)
  StateHistory[06] = DeviceNodeStartPostWork (0x307)
  StateHistory[05] = DeviceNodeStartCompletion (0x306)
  StateHistory[04] = DeviceNodeStartPending (0x305)
  ...
  Flags (0x6c000131)  DNF_MADEUP, DNF_ENUMERATED,
                      DNF_IDS_QUERIED, DNF_NO_RESOURCE_REQUIRED,
                      DNF_NO_LOWER_DEVICE_FILTERS, DNF_NO_LOWER_CLASS_FILTERS,
                      DNF_NO_UPPER_DEVICE_FILTERS, DNF_NO_UPPER_CLASS_FILTERS
  UserFlags (0x00000008)  DNUF_NOT_DISABLEABLE
  DisableableDepends = 1 (including self)
```

!devobj

Displays detailed information about a DEVICE_OBJECT.

Here is an example:

```
kd> !devobj 0xff0d4af0

Device object (ff0d4af0) is for:
 00252d \Driver\PnpManager DriverObject ff0d9030
Current Irp 00000000 RefCount 0 Type 00000004 Flags 00001040AttachedDev ff0b59e0

DevExt ff0d4ba8 DevNode ff0d4a08
Device queue is not busy.
```

!drivers

The **!drivers** command is no longer supported. Use the **lm t n** command instead.

!drvobj

Displays detailed information about a DRIVER_OBJECT.

Lists all the device objects created by the specified driver.

Here is an example:

```
kd> !drvobj serial

Driver object (ff0ba630) is for:
 \Driver\Serial
Driver Extension List: (id , addr)

Device Object list:
ffba3040  ff0b4040  ff0b59e0  ff0b5040
```

!ecb, !ecd, !ecw

(x86 target computers only) Writes a sequence of values into the PCI configuration space.

ib, iw, id

Reads data from an I/O port.

These three commands are useful for determining whether a certain I/O range is claimed by a device other than

the driver being debugged. A byte value of 0xFF at a port indicates that the port is not in use.

!ioreslist

Displays the specified IO_RESOURCE_REQUIREMENTS_LIST.

!irp

Displays information about an IRP.

!irpfind

Displays information about all IRPs currently allocated in the target system, or information about those IRPs whose fields match the specified search criteria.

!pci

(x86 target computers only) Displays the current status of the PCI buses and any devices attached to them. It can also display the PCI configuration space.

The following example displays the devices on the primary bus:

```
kd> !pci
PCI Bus 0
00:0  8086:1237.02  Cmd[0106:.mb..s]  Sts[2280:.....]  Device  Host bridge
0d:0  8086:7000.01  Cmd[0007:imb...]  Sts[0280:.....]  Device  ISA bridge
0d:1  8086:7010.00  Cmd[0005:i.b...]  Sts[0280:.....]  Device  IDE controller
0e:0  1011:0021.01  Cmd[0107:imb..s]  Sts[0280:.....]  PciBridge 0->1-1  PCI-PCI
 bridge
10:0  5333:8811.43  Cmd[0023:im.v..]  Sts[0200:.....]  Device  VGA compatible controller



The following example displays the devices for the secondary bus, with verbose output:

kd> !pci 1 1

PCI Bus 1
08:0  10b7:5900.00  Cmd[0107:imb..s]  Sts[0200:.....]  Device  Ethernet
      cf8:80014000  IntPin:1  IntLine:f  Rom:fa000000  cis:0  cap:0
      IO[0]:fce1

09:0  9004:8178.00  Cmd[0117:imb..s]  Sts[0280:.....]  Device  SCSI controller
      cf8:80014800  IntPin:1  IntLine:f  Rom:fa000000  cis:0  cap:0
      IO[0]:f801       MEM[1]:f9fff000

0b:0  9004:5800.10  Cmd[0116:.mb..s]  Sts[0200:.....]  Device  SubID:9004:8940
1394 host controller
      cf8:80015800  IntPin:1  IntLine:e  Rom:fa000000  cis:0  cap:0
      MEM[0]:f9ffec00
```

The following example displays the PCI configuration space for the SCSI controller (bus 1, device 9, function 0):

```
kd> !pci 100 1 9 0
00: 9004    ;VendorID=9004
02: 8178    ;DeviceID=8178
04: 0117    ;Command=SERREnable,MemWriteEnable,BusInitiate,MemSpaceEnable,IOSpac
eEnable
06: 0280    ;Status=FB2BCapable,DEVSELTiming:1
08: 00      ;RevisionID=00
09: 00      ;ProgIF=00 (SCSI bus controller)
0a: 00      ;SubClass=00
0b: 01      ;BaseClass=01 (Mass storage controller)
0c: 08      ;CacheLineSize=Burst8DW
0d: 20      ;LatencyTimer=20
0e: 00      ;HeaderType=00
0f: 00      ;BIST=00
10: 0000f801;BAR0=0000f801
14: f9fff000;BAR1=f9fff000
18: 00000000;BAR2=00000000
1c: 00000000;BAR3=00000000
20: 00000000;BAR4=00000000
24: 00000000;BAR5=00000000
28: 00000000;CBCISPtr=00000000
2c: 0000    ;SubSysVenID=0000
2e: 0000    ;SubSysID=0000
30: fa000000;ROMBAR=fa000000
34: 00000000;Reserved=00000000
38: 00000000;Reserved=00000000
3c: 0f      ;IntLine=0f
3d: 01      ;IntPin=01
3e: 08      ;MinGnt=08
3f: 08      ;MaxLat=08
40: 00001580,00001580,00000000,00000000,00000000,00000000,00000000,00000000
60: 00000000,00000000,00000000,00000000,00000000,00000000,00000000,00000000
80: 00000000,00000000,00000000,00000000,00000000,00000000,00000000,00000000
a0: 00000000,00000000,00000000,00000000,00000000,00000000,00000000,00000000
c0: 00000000,00000000,00000000,00000000,00000000,00000000,00000000,00000000
e0: 00000000,00000000,00000000,00000000,00000000,00000000,00000000,00000000
```

!pcitree

Displays information about PCI device objects, including child PCI buses and CardBus buses, as well as the devices attached to them.

!pnpevent

Displays the PnP device event queue.

!rellist

Displays a PnP relation list and any related CM_RESOURCE_LIST and IO_RESOURCE_LIST structures.

Determining the Status of a Device

12/5/2018 • 3 minutes to read • Edit Online

To display the entire device tree, starting with the root, use **!devnode 0 1**:

```
kd> !devnode 0 1
```

Identify the devnode for which you are searching, either by examining the driver or the bus that exposed it.

The devnode status flags describe the status of the device. For details, see Device Node Status Flags.

In the example for the **!devnode** command in the Extensions for Debugging Plug and Play Drivers section, the swenum device was created by the PnP Manager, so the DNF_MADEUP (0x00000001) flag is set.

The following example shows a device that was created by the PCI bus. This device does not have the DNF_MADEUP flag set.

```
0: kd> !devnode 0xfffffa8004483490
DevNode 0xfffffa8004483490 for PDO 0xfffffa800448d060
  Parent 0xfffffa80036766d0   Sibling 0xfffffa8004482010   Child 0xfffffa80058ad720
  InstancePath is "PCI\VEN_8086&DEV_293C&SUBSYS_2819103C&REV_02\3&21436425&0&D7"
  ServiceName is "usbehci"
  State = DeviceNodeStarted (0x308)
  Previous State = DeviceNodeEnumerateCompletion (0x30d)
  StateHistory[09] = DeviceNodeEnumerateCompletion (0x30d)
  StateHistory[08] = DeviceNodeEnumeratePending (0x30c)
  StateHistory[07] = DeviceNodeStarted (0x308)
  StateHistory[06] = DeviceNodeStartPostWork (0x307)
  ...
  Flags (0x6c0000f0)  DNF_ENUMERATED, DNF_IDS_QUERIED,
                      DNF_HAS_BOOT_CONFIG, DNF_BOOT_CONFIG_RESERVED,
                      DNF_NO_LOWER_DEVICE_FILTERS, DNF_NO_LOWER_CLASS_FILTERS,
                      DNF_NO_UPPER_DEVICE_FILTERS, DNF_NO_UPPER_CLASS_FILTERS
  CapabilityFlags (0x00002000)  WakeFromD3
```

Examples

1. A devnode for a device with insufficient resources:

```
kd> !devnode 0xff0d06e8 6

DevNode 0xff0d06e8 for PDO 0xff0d07d0 at level 0x3
  Parent 0xff0d1408   Sibling 0000000000   Child 0000000000
  InterfaceType 0xffffffff  Bus Number 0xfffffff0
  InstancePath is "ISAPNP\SUP2171\00000067"
  ServiceName is "Modem"
  TargetDeviceNotify List - f 0xff0d074c  b 0xff0d074c
  Flags (..........)  DNF_PROCESSED, DNF_ENUMERATED,
                      DNF_INSUFFICIENT_RESOURCES, DNF_ADDED,
                      DNF_HAS_BOOT_CONFIG
                      Unknown flags 0x40000000

  IoResList at 0xe133e7a8 : Interface 0x1  Bus 0  Slot 0
    Alternative 0 (Version 1.1)
      Preferred Descriptor 0 - NonArbitrated/ConfigData (0x80) Shared (0x3)
        Flags (0000) -
        Data:              : 0x0 0x61004d 0x680063
      Preferred Descriptor 1 - Port (0x1) Undetermined Sharing (0)
        Flags (0x11) - PORT_IO 16_BIT_DECODE
        0x000008 byte range with alignment 0x000001
        2f8 - 0x2ff
      Preferred Descriptor 2 - Interrupt (0x2) Shared (0x3)
        Flags (0x01) - LATCHED
        0x3 - 0x3
```

Note that the devnode has no CM Resource List, because it is not started and is not using resources, although it has requested resources.

2. Note that there are no resources stored in this devnode for a legacy driver.

```
kd> !devnode 0xff0d1648 6

DevNode 0xff0d1648 for PDO 0xff0d22d0 at level 0x2
  Parent 0xff0d2e28   Sibling 0xff0d1588   Child 0000000000
  InterfaceType 0xffffffff  Bus Number 0xffffffff
  InstancePath is "PCI\VEN_102B&DEV_0519\0&60"
  ServiceName is "mga_mil"
  TargetDeviceNotify List - f 0xff0d16ac  b 0xff0d16ac
  Flags (0x6000500b)  DNF_PROCESSED, DNF_STARTED,
                      DNF_ENUMERATED, DNF_ADDED,
                      DNF_LEGACY_DRIVER, DNF_HAS_BOOT_CONFIG
                      Unknown flags 0x40000000
```

You can retrieve the device object list for the driver for the following types of devices:

```
kd> !drvobj mga_mil

Driver object (ff0bbc10) is for:
 \Driver\mga_mil
Driver Extension List: (id , addr)

Device Object list:
ff0bb900
```

You can then dump the data for this device object:

```
kd> !devobj ff0bb900

Device object (ff0bb900) is for:
 Video0 \Driver\mga_mil DriverObject ff0bbc10
Current Irp 00000000 RefCount 1 Type 00000023 Flags 0000204c
DevExt ff0bb9b8 DevNode ff0bb808
Device queue is not busy.
```

Finally, you can dump the devnode referred by the device object. This devnode is not linked in the device tree. It represents a "pseudo-devnode" used to claim resources for the legacy device. Note the DNF_RESOURCE_REPORTED flag that indicates the device is a reported detected device.

```
kd> !devnode ff0bb808 6

DevNode 0xff0bb808 for PDO 0xff0bb900 at level 0xffffffff
  Parent 0xff0daf48   Sibling 0000000000   Child 0000000000
  InterfaceType 0xffffffff  Bus Number 0xffffffff
  TargetDeviceNotify List - f 0xff0bb86c  b 0xff0bb86c
  Flags (0x00000400)  DNF_RESOURCE_REPORTED
  CmResourceList at 0xe12474e8  Version 0.0  Interface 0x5  Bus #0
    Entry 0 - Port (0x1) Shared (0x3)
      Flags (0x01) - PORT_MEMORY PORT_IO
      Range starts at 0x3c0 for 0x10 bytes
    Entry 1 - Port (0x1) Shared (0x3)
      Flags (0x01) - PORT_MEMORY PORT_IO
      Range starts at 0x3d4 for 0x8 bytes
    Entry 2 - Port (0x1) Shared (0x3)
      Flags (0x01) - PORT_MEMORY PORT_IO
      Range starts at 0x3de for 0x2 bytes
    Entry 3 - Memory (0x3) Device Exclusive (0x1)
      Flags (0000) - READ_WRITE
      Range starts at 0x0000000040000000 for 0x4000 bytes
    Entry 4 - Memory (0x3) Device Exclusive (0x1)
      Flags (0000) - READ_WRITE
      Range starts at 0x0000000040800000 for 0x800000 bytes
```

Device Node Status Flags

12/5/2018 • 2 minutes to read • Edit Online

The Device Node Status flags describe the status of a device.

The most important flags are:

DNF_MADEUP (0x00000001)
The device was created and is owned by the PnP Manager. It was not created by a bus driver.

DNF_DUPLICATE (0x00000002)
The device node is a duplicate of another enumerated device node.

DNF_HAL_NODE (0x00000004)
The device node is the root node created by the hardware abstraction layer (HAL).

DNF_REENUMERATE (0x00000008)
The device needs to be re-enumerated.

DNF_ENUMERATED (0x00000010)
The PDO for the device was exposed by its parent.

DNF_IDS_QUERIED (0x00000020)
The operating system should send IRP_MN_QUERY_ID requests to the device driver.

DNF_HAS_BOOT_CONFIG (0x00000040)
The device has resources assigned by the BIOS. The device is considered pseudo-started and needs to participate in rebalancing.

DNF_BOOT_CONFIG_RESERVED (0x00000080)
The boot resources of the device are reserved.

DNF_NO_RESOURCE_REQUIRED (0x00000100)
The device does not require resources.

DNF_RESOURCE_REQUIREMENTS_NEED_FILTERED (0x00000200)
The device's resource requirements list is a filtered list.

DNF_RESOURCE_REQUIREMENTS_CHANGED (0x00000400)
The device's resource requirements list has changed.

DNF_NON_STOPPED_REBALANCE (0x00000800)
The device can be restarted with new resources without being stopped.

DNF_LEGACY_DRIVER (0x00001000)
The device's controlling driver is a non-PnP driver.

DNF_HAS_PROBLEM (0x00002000)
The device has a problem and will be removed.

DNF_HAS_PRIVATE_PROBLEM (0x00004000)
The device reported PNP_DEVICE_FAILED without also reporting PNP_DEVICE_RESOURCE_REQUIREMENTS_CHANGED.

DNF_HARDWARE_VERIFICATION (0x00008000)
The device node has hardware verification.

DNF_DEVICE_GONE (0x00010000)
The device's PDO is no longer returned in an IRP_QUERY_RELATIONS request.

DNF_LEGACY_RESOURCE_DEVICENODE (0x00020000)
The device node was created for legacy resource allocation.

DNF_NEEDS_REBALANCE (0x00040000)
The device node has triggered rebalancing.

DNF_LOCKED_FOR_EJECT (0x00080000)
The device is being ejected or is related to a device that is being ejected.

DNF_DRIVER_BLOCKED (0x00100000)
One or more of the drivers for the device node have been blocked from loading.

DNF_CHILD_WITH_INVALID_ID (0x00200000)
One or more children of the device node have invalid IDs.

DNF_ASYNC_START_NOT_SUPPORTED (0x00400000)
The device does not support asynchronous starts.

DNF_ASYNC_ENUMERATION_NOT_SUPPORTED (0x00800000)
The device does not support asynchronous enumeration.

DNF_LOCKED_FOR_REBALANCE (0x01000000)
The device is locked for rebalancing.

DNF_UNINSTALLED (0x02000000)
An IRP_MN_QUERY_REMOVE_DEVICE request is in progress for the device.

DNF_NO_LOWER_DEVICE_FILTERS (0x04000000)
There is no Registry entry of the lower-device-filters type for the device.

DNF_NO_LOWER_CLASS_FILTERS (0x08000000)
There is no Registry entry of the lower-class-filters type for the device.

DNF_NO_SERVICE (0x10000000)
There is no Registry entry of the service the for the device.

DNF_NO_UPPER_DEVICE_FILTERS (0x20000000)
There is no Registry entry of the upper-device-filters type for the device.

DNF_NO_UPPER_CLASS_FILTERS (0x40000000)
There is no Registry entry of the upper-class-filters type for the device.

DNF_WAITING_FOR_FDO (0x80000000)
Enumeration of the device is waiting until the driver attaches its FDO.

Device Manager Problem Codes

12/5/2018 • 2 minutes to read • Edit Online

The Device Manager marks a device with a yellow exclamation mark (!) when the device has a problem. The problem codes are in the form CM_PROB_*XXX* and are defined in the header file cfg.h. The most important are explained here, together with their mapping to the Device Node Status Flags. For a more comprehensive list, please see Device Manager Error Messages.

Code 1 (CM_PROB_NOT_CONFIGURED)
Indicates that the device is not installed and was not installed previously. (Corresponds to DNF_NOT_CONFIGURED.)

Code 10 (CM_PROB_FAILED_START)
Indicates that the device did not start for some reason, but the I/O Manager attempted to start it with a set of resources. (Corresponds to DNF_START_FAILED.)

Code 12 (CM_PROB_NORMAL_CONFLICT)
Indicates that there were not sufficient resources to start this device. (Corresponds to DNF_INSUFFICIENT_RESOURCES.)

Code 14 (CM_PROB_NEED_RESTART)
Indicates that user mode reconfigured the device and a reboot is required for the changes to take effect. (Corresponds to DNF_NEED_RESTART.)

Code 18 (CM_PROB_REINSTALL)
Indicates that the device needs to be installed and was installed previously. (Corresponds to DNF_REINSTALL.)

Code 21 (CM_PROB_WILL_BE_REMOVED)
Indicates that the user mode uninstalled this device. (Corresponds to DNF_WILL_BE_REMOVED.)

Code 22 (CM_PROB_DISABLED)
Indicates that the device is disabled. (Corresponds to DNF_DISABLED.)

Code 28 (CM_PROB_FAILED_INSTALL)
Indicates that the installation failed and there is no driver selected for this device, although the kernel did not report a problem (and there is no DNF_XXX match for this the problem). This problem can be the result of an on-board system device (ISA timer, ISA RTC, RAM Memory, and so forth) that does not yet have an INF file.

Code 31 (CM_FAILED_ADD)
Indicates that the device was not added. Reasons for the failure may include: a driver's **AddDevice** routine returned an error, there is no service listed for the device in the registry, there is more than one service listed, or the controlling driver could not be loaded. (Corresponds to DNF_ADD_FAILED.)

Checking for Resource Conflicts

12/5/2018 • 11 minutes to read • Edit Online

This section discusses techniques that can be used to detect resource conflicts.

The first technique involves dumping the arbiter data. The following example examines the arbiter data for the I/O ranges:

```
kd> !arbiter 1

DEVNODE ff0daf48
  Port Arbiter "RootPort" at 8045b920
    Allocated ranges:
      10 - 1f S         Owner ff0d6b30
      22 - 3f S         Owner ff0d6b30
      44 - 47 S         Owner ff0d6b30
      4c - 6f S         Owner ff0d6b30
      72 - 7f S         Owner ff0d6b30
      90 - 91 S         Owner ff0d6b30
      93 - 9f S         Owner ff0d6b30
      a2 - bf S         Owner ff0d6b30
      d0 - ef S         Owner ff0d6b30
      100 - 2f7 S       Owner ff0d6b30
      300 - cf7 S       Owner ff0d6b30
      d00 - ffff S      Owner ff0d6b30
    Possible allocation:
      < none >

    DEVNODE ff0d2e28 (PCI_HAL\PNP0A03\0)
      Port Arbiter "PCI I/O Port (b=00)" at e122c2c8
        Allocated ranges:
          0 - f         Owner 00000000
          20 - 21       Owner 00000000
          40 - 43       Owner 00000000
          48 - 4b       Owner 00000000
          60 - 60       Owner ff0d4030
          64 - 64       Owner ff0d4030
          70 - 71       Owner 00000000
          80 - 8f       Owner 00000000
          92 - 92       Owner 00000000
          a0 - a1       Owner 00000000
          c0 - cf       Owner 00000000
          f0 - ff       Owner 00000000
          170 - 177     Owner ff0cf030
          1ce - 1cf S   Owner ff040040
          2f8 - 2ff     Owner 00000000
          376 - 376     Owner ff0cf030
          378 - 37f     Owner ff0d4e70
          3b0 - 3bb S   Owner ff040040
 3c0 - 3cf S   Owner ff0bb900
          3c0 - 3df S   Owner ff040040
          3d4 - 3db S   Owner ff0bb900
          3de - 3df S   Owner ff0bb900
 3ec - 3ef     Owner ff0d0b50 (This device conflicts with another device, see below)
          3f2 - 3f5     Owner ff0d4770
          3f7 - 3f7 S   Owner ff0d4770
          3f8 - 3ff     Owner ff0d4af0
          778 - 77b     Owner ff0d4e70
          cf8 - cff     Owner 00000000
          1000 - 10ff           Owner ff0d1030
          1400 - 140f           Owner ff0d1d30
          1410 - 141f           Owner ff0d1890
          10000 - ffffffffffffffff      Owner 00000000
        Possible allocation:
          < none >
```

Note that there are two arbiters: one located in the root of the device tree, and one in PCI_HAL. Also note that the PCI arbiter claims and preallocates ranges for the devices it arbitrates (0xD000-0xFFFF, which is later suballocated by the PCI arbiter for its devices). The Owner field indicates the device object that owns the range. A value of zero for Owner indicates that the range is not on the bus. In the case of a PCI bridge, for example, all the ranges it does not pass will be assigned to **NULL**.

In the following example, the PCI bridge passes I/O 0xD000-0xDFFFF so its arbiter will contain the following two

ranges:

```
0-CFFFF            Owner 00000000
E0000-FFFFFFFFFFFFFFFF   Owner 00000000
```

The FFFFFFFFFFFFFFFF is because all arbitrated resources are treated as 64-bit ranges.

Examples:

```
kd> !devobj ff0bb900

Device object (ff0bb900) is for:
 Video0 \Driver\mga_mil DriverObject ff0bbc10
Current Irp 00000000 RefCount 1 Type 00000023 Flags 0000204c
DevExt ff0bb9b8 DevNode ff0bb808
Device queue is not busy.
kd> !devnode ff0bb808

DevNode 0xff0bb808 for PDO 0xff0bb900 at level 0xffffffff
  Parent 0xff0daf48   Sibling 0000000000   Child 0000000000
  InterfaceType 0xffffffff  Bus Number 0xffffffff
  TargetDeviceNotify List - f 0xff0bb86c  b 0xff0bb86c
  Flags (0x00000400)  DNF_RESOURCE_REPORTED
kd> !devnode ff0bb808 6

DevNode 0xff0bb808 for PDO 0xff0bb900 at level 0xffffffff
  Parent 0xff0daf48   Sibling 0000000000   Child 0000000000
  InterfaceType 0xffffffff  Bus Number 0xffffffff
  TargetDeviceNotify List - f 0xff0bb86c  b 0xff0bb86c
  Flags (0x00000400)  DNF_RESOURCE_REPORTED
  CmResourceList at 0xe12474e8  Version 0.0  Interface 0x5  Bus #0
    Entry 0 - Port (0x1) Shared (0x3)
      Flags (0x01) - PORT_MEMORY PORT_IO
      Range starts at 0x3c0 for 0x10 bytes
    Entry 1 - Port (0x1) Shared (0x3)
      Flags (0x01) - PORT_MEMORY PORT_IO
      Range starts at 0x3d4 for 0x8 bytes
    Entry 2 - Port (0x1) Shared (0x3)
      Flags (0x01) - PORT_MEMORY PORT_IO
      Range starts at 0x3de for 0x2 bytes
    Entry 3 - Memory (0x3) Device Exclusive (0x1)
      Flags (0000) - READ_WRITE
      Range starts at 0x0000000040000000 for 0x4000 bytes
    Entry 4 - Memory (0x3) Device Exclusive (0x1)
      Flags (0000) - READ_WRITE
      Range starts at 0x0000000040800000 for 0x800000 bytes
```

As shown in the example, this operation retrieved the legacy video card that owns the range 3c0-3cf. The same device object is listed near the other ranges it owns (3de-3df and 3d4-3dc). Using the same tracking technique, the range of 3f8-3ff is determined to be that used by the serial port.

A similar technique is required to translate the interrupts:

```
kd> !arbiter 4

DEVNODE ff0daf48
  Interrupt Arbiter "RootIRQ" at 8045bae0
    Allocated ranges:
      31 - 31           Owner ff0d4030
      34 - 34 S         Owner ff0d4af0
      36 - 36           Owner ff0d4770
      3b - 3b S         Owner ff0d1030
      3b - 3b S         Owner ff0d1d30
      3c - 3c           Owner ff0d3c70
 3f - 3f           Owner ff0cf030
    Possible allocation:
      < none >
```

Note that there is a single arbiter for interrupts: the root arbiter.

For example, translate the interrupt 3F to an IRQ. First dump the device object, then the devnode:

```
kd> !devobj ff0cf030

Device object (ff0cf030) is for:
 IdeFdoff0d0398Channel1 \Driver\IntelIde DriverObject ff0d0530
Current Irp 00000000 RefCount 0 Type 00000004 Flags 00001040AttachedDev ff0cd030

DevExt ff0cf0e8 DevNode ff0cfe88
Device queue is not busy.
kd> !devnode ff0cfe88 6

DevNode 0xff0cfe88 for PDO 0xff0cf030 at level 0x3
  Parent 0xff0d1348   Sibling 0000000000   Child 0xff0c84a8
  InterfaceType 0xffffffff  Bus Number 0xfffffff0
  InstancePath is "PCIIDE\IDEChannel\1&1"
  ServiceName is "atapi"
  TargetDeviceNotify List - f 0xff0cfeec  b 0xff0cfeec
  Flags (0x6000120b)  DNF_PROCESSED, DNF_STARTED,
                      DNF_ENUMERATED, DNF_RESOURCE_ASSIGNED,
                      DNF_ADDED, DNF_HAS_BOOT_CONFIG
                      Unknown flags 0x40000000
  CmResourceList at 0xe12321c8  Version 0.0  Interface 0x1  Bus #0
    Entry 0 - Port (0x1) Device Exclusive (0x1)
      Flags (0x01) - PORT_MEMORY PORT_IO
      Range starts at 0x170 for 0x8 bytes
    Entry 1 - Port (0x1) Device Exclusive (0x1)
      Flags (0x01) - PORT_MEMORY PORT_IO
      Range starts at 0x376 for 0x1 bytes
    Entry 2 - Interrupt (0x2) Device Exclusive (0x1)
      Flags (0x01) - LATCHED
      Level 0xf, Vector 0xf, Affinity 0xffffffff

  IoResList at 0xe12363c8 : Interface 0x1  Bus 0  Slot 0
  Reserved Values = {0x0002e0d0, 0x00920092, 0xe1235508}
    Alternative 0 (Version 1.1)
      Preferred Descriptor 0 - NonArbitrated/ConfigData (0x80) Shared (0x3)
        Flags (0000) -
        Data:              : 0x1 0x61004d 0x680063
      Preferred Descriptor 1 - Port (0x1) Device Exclusive (0x1)
        Flags (0x01) - PORT_IO
        0x000008 byte range with alignment 0x000001
        170 - 0x177
      Preferred Descriptor 2 - Port (0x1) Device Exclusive (0x1)
        Flags (0x01) - PORT_IO
        0x000001 byte range with alignment 0x000001
        376 - 0x376
      Preferred Descriptor 3 - Interrupt (0x2) Device Exclusive (0x1)
        Flags (0x01) - LATCHED
        0xf - 0xf
    Alternative 1 (Version 1.1)
      Preferred Descriptor 0 - Port (0x1) Device Exclusive (0x1)
        Flags (0x01) - PORT_IO
        0x000008 byte range with alignment 0x000001
        170 - 0x177
      Preferred Descriptor 1 - Port (0x1) Device Exclusive (0x1)
        Flags (0x01) - PORT_IO
        0x000001 byte range with alignment 0x000001
        376 - 0x376
      Preferred Descriptor 2 - Interrupt (0x2) Device Exclusive (0x1)
        Flags (0x01) - LATCHED
        0xf - 0xf
```

For example, try to determine if there is a resource conflict that caused this device not to start, starting with a **devnode**:

```
kd> !devnode 0xff0d4bc8 6

DevNode 0xff0d4bc8 for PDO 0xff0d4cb0 at level 0
  Parent 0xff0daf48   Sibling 0xff0d4a08   Child 0000000000
  InterfaceType 0xffffffff  Bus Number 0xffffffff
  InstancePath is "Root\*PNP0501\1_0_17_2_0_0"
  ServiceName is "Serial"
  TargetDeviceNotify List - f 0xff0d4c2c  b 0xff0d4c2c
  Flags (0x60001129)  DNF_PROCESSED, DNF_ENUMERATED,
                      DNF_MADEUP, DNF_INSUFFICIENT_RESOURCES,
                      DNF_ADDED, DNF_HAS_BOOT_CONFIG
                      Unknown flags 0x40000000

  IoResList at 0xe1251e28 : Interface 0x1  Bus 0  Slot 0
  Reserved Values = {0x0043005c, 0x006e006f, 0x00720074}
    Alternative 0 (Version 1.1)
      Preferred Descriptor 0 - NonArbitrated/ConfigData (0x80) Undetermined Shar
ing (0)
        Flags (0000) -
        Data:              : 0xc000 0x0 0x0
      Preferred Descriptor 1 - Port (0x1) Undetermined Sharing (0)
        Flags (0x05) - PORT_IO 10_BIT_DECODE
        0x000008 byte range with alignment 0x000001
        3e8 - 0x3ef
      Preferred Descriptor 2 - Interrupt (0x2) Shared (0x3)
        Flags (0x01) - LATCHED
        0x5 - 0x5
    Alternative 1 (Version 1.1)
      Preferred Descriptor 0 - NonArbitrated/ConfigData (0x80) Undetermined Shar
ing (0)
        Flags (0000) -
        Data:              : 0xc000 0x0 0x0
      Preferred Descriptor 1 - Port (0x1) Undetermined Sharing (0)
        Flags (0x05) - PORT_IO 10_BIT_DECODE
        0x000008 byte range with alignment 0x000008
        3e8 - 0x3ef
      Preferred Descriptor 2 - Interrupt (0x2) Shared (0x3)
        Flags (0x01) - LATCHED
        0x5 - 0x5
```

First, make the assumption that this is an I/O conflict and dump the arbiters (see the preceding example). The result shows that the range 0x3EC-0x3EF is owned by 0xFF0D0B50, which overlaps the serial device's resources request. Next, dump the device object for the owner of this range, and then dump the devnode for the owner:

```
kd> !devobj ff0d0b50

Device object (ff0d0b50) is for:
 Resource00413e \Driver\isapnp DriverObject ff0d0e10
Current Irp 00000000 RefCount 0 Type 00000004 Flags 00001040
DevExt ff0d0c08 DevNode ff0d0a68
Device queue is not busy.
kd> !devnode ff0d0a68 6

DevNode 0xff0d0a68 for PDO 0xff0d0b50 at level 0xffffffff
  Parent 0xff0daf48   Sibling 0000000000   Child 0000000000
  InterfaceType 0xffffffff  Bus Number 0xffffffff
Duplicate PDO 0xff0d0e10  TargetDeviceNotify List - f 0xff0d0acc  b 0xff0d0acc
  Flags (0x00000421)  DNF_PROCESSED, DNF_MADEUP,
                      DNF_RESOURCE_REPORTED
  CmResourceList at 0xe1233628  Version 0.0  Interface 0x1  Bus #0
    Entry 0 - Port (0x1) Device Exclusive (0x1)
      Flags (0x01) - PORT_MEMORY PORT_IO
      Range starts at 0x3ec for 0x4 bytes
```

This is a "pseudo-devnode" that corresponds to the range allocated by the ISAPNP driver for its read data port.

To determine the resources that the PnP Manager assigned to a particular device when it attempted to start the device:

1. Place a breakpoint on the routine that is called when the IRP_MN_START_DEVICE request is received by the driver. You can also place a breakpoint on the driver's dispatch routine (if you know its name). In both cases, the driver and its symbols should be loaded. This may require you to set an initial breakpoint.

   For example, for PCMCIA you can set a breakpoint on **pcmcia!pcmciastartpccard**. The advantage of using this particular routine is that its second parameter is a CM Resource List that you can dump using **!cmreslist** (eliminating step 3). See the following PCMCIA example.

2. When you have determined which device is of interest, dump the device object (if you have not done so already), and then dump the devnode with the CM Resource List. Check which resources were assigned to the device. You may also check whether the resources are a subset of the I/O Resource List. Then type **g** or single step through the procedure, and determine whether the device was started and which resources were assigned. If a device was offered a set of resources to start but failed to do so, the driver might not be behaving properly (for example, if it incorrectly declared that it can use a set of resources it cannot actually use).

Example:

```
ntoskrnl!IopStartDevice:
80420212 55              push    ebp
kd> kb
ChildEBP RetAddr  Args to Child
f64138c0 8048b640 ff0ce870 ff0d7c08 ff0cde88 ntoskrnl!IopStartDevice (!devobj ff0ce870)
f64138f0 8048d8e7 ff0cde88 f6413978 ff0cddc8 ntoskrnl!IopStartAndEnumerateDevice
+0x1a
f6413900 8048de7f ff0cde88 f6413978 ff0cf448 ntoskrnl!IopProcessStartDevicesWork
er+0x43
f6413910 8048d8d5 ff0cf448 8048d8a4 f6413978 ntoskrnl!IopForAllChildDeviceNodes+
0x1f
f6413924 8048de7f ff0cf448 f6413978 ff0d3f48 ntoskrnl!IopProcessStartDevicesWork
er+0x31
f6413934 8048d8d5 ff0d3f48 8048d8a4 f6413978 ntoskrnl!IopForAllChildDeviceNodes+
0x1f
f6413948 8048d893 ff0d3f48 f6413978 e12052e8 ntoskrnl!IopProcessStartDevicesWork
er+0x31
f641395c 804f6f1b ff0d7c08 f6413978 8045c520 ntoskrnl!IopProcessStartDevices+0x1
f
f64139d0 804f5cc1 80088000 f6413aec 8045bba0 ntoskrnl!IopInitializeBootDrivers+0
x2f9
f6413b24 804f4db3 80088000 00000001 00000000 ntoskrnl!IoInitSystem+0x3a6
f6413da8 80447610 80088000 00000000 00000000 ntoskrnl!Phase1Initialization+0x6a3

f6413ddc 8045375a 804f4710 80088000 00000000 ntoskrnl!PspSystemThreadStartup+0x5
4
00000000 00000000 00000000 00000000 00000000 ntoskrnl!KiThreadStartup+0x16
kd> !devobj ff0ce870

Device object (ff0ce870) is for:
 NTPNP_PCI0002 \Driver\PCI DriverObject ff0ceef0
Current Irp 00000000 RefCount 0 Type 00000022 Flags 00001040AttachedDev ff0cb9e0

DevExt ff0ce928 DevNode ff0cde88
Device queue is not busy.
kd> !devnode ff0cde88 6

DevNode 0xff0cde88 for PDO 0xff0ce870 at level 0x2
  Parent 0xff0cf448   Sibling 0xff0cddc8   Child 0000000000
  InterfaceType 0xffffffff  Bus Number 0xffffffff
  InstancePath is "PCI\VEN_8086&DEV_7010\0&69"
  ServiceName is "intelide"
  TargetDeviceNotify List - f 0xff0cdeec  b 0xff0cdeec
  Flags (0x00001209)  DNF_PROCESSED, DNF_ENUMERATED,
                      DNF_RESOURCE_ASSIGNED, DNF_ADDED
 (note that the device is not yet started)
  CmResourceList at 0xe120fce8  Version 0.0  Interface 0x5  Bus #0
    Entry 0 - Port (0x1) Device Exclusive (0x1)
      Flags (0x01) - PORT_MEMORY PORT_IO
      Range starts at 0xfff0 for 0x10 bytes (these are the resources used: 0xfff0-0xffff)
    Entry 1 - DevicePrivate (0x81) Device Exclusive (0x1)
      Flags (0000) -
      Data - {0x00000001, 0x00000004, 0000000000}

  IoResList at 0xe120df88 : Interface 0x5  Bus 0  Slot 0x2d
    Alternative 0 (Version 1.1)
      Descriptor 0 - Port (0x1) Device Exclusive (0x1)
        Flags (0x01) - PORT_IO
        0x000010 byte range with alignment 0x000010
 0 - 0xffff     (it could have used any 16 bytes that are 16-byte aligned between 0 and 0xffff)
      Descriptor 1 - DevicePrivate (0x81) Device Exclusive (0x1)
        Flags (0000) -
        Data:              : 0x1 0x4 0x0
```

Example for PCMCIA:

```
kd> bp pcmcia!pcmciastartpccard
Loading symbols for 0x8039d000       pcmcia.sys ->   pcmcia.sys
kd> kb
Loading symbols for 0x80241000         ndis.sys ->   ndis.sys
ChildEBP RetAddr  Args to Child
f6413814 803a7cbd ff0d0c30 e11d8808 ff0d0c30 pcmcia!PcmciaStartPcCard
f6413838 803a3798 ff0d0c30 ff0d1500 ff0d1588 pcmcia!PcmciaPdoPnpDispatch+0x169
f6413848 80418641 ff0d0c30 ff0d1588 00000000 pcmcia!PcmciaDispatch+0x3a
f641385c 802455cf ff0d1614 ff0d1638 00040000 ntoskrnl!IofCallDriver+0x35
f641387c 802497cf ff0d1588 ff0d0c30 ff0c8210 NDIS!ndisPassIrpDownTheStack+0x3b
f64138ac 80418641 ff0c8210 ff0d161c ff0d1640 NDIS!ndisPnPDispatch+0x1f9
f64138c0 8048de68 ff0c3508 ff0d16a8 00000000 ntoskrnl!IofCallDriver+0x35
f64138d4 8041ff5e ff0c8210 f64138f8 ff0c3508 ntoskrnl!IopAsynchronousCall+0x90
f6413920 8048b4ae ff0d0c30 ff0e0a68 ff0d16a8 ntoskrnl!IopStartDevice+0x76
f6413950 8048d707 ff0d16a8 f64139fc 00000000 ntoskrnl!IopStartAndEnumerateDevice+0x1a
f6413960 8048dc9f ff0d16a8 f64139fc ff0d1e88 ntoskrnl!IopProcessStartDevicesWorker+0x43
f6413970 8048d6f5 ff0d1e88 8048d6c4 f64139fc ntoskrnl!IopForAllChildDeviceNodes+0x1f
f6413984 8048dc9f ff0d1e88 f64139fc ff0d3268 ntoskrnl!IopProcessStartDevicesWorker+0x31
f6413994 8048d6f5 ff0d3268 8048d6c4 f64139fc ntoskrnl!IopForAllChildDeviceNodes+0x1f
f64139a8 8048dc9f ff0d3268 f64139fc ff0d7b28 ntoskrnl!IopProcessStartDevicesWorker+0x31
f64139b8 8048d6f5 ff0d7b28 8048d6c4 f64139fc ntoskrnl!IopForAllChildDeviceNodes+0x1f
f64139cc 8048d6b3 ff0d7b28 f64139fc 80087000 ntoskrnl!IopProcessStartDevicesWorker+0x31
f64139e0 804f6c97 ff0e0a68 f64139fc 8045c140 ntoskrnl!IopProcessStartDevices+0x1f
f6413a30 804f5601 000001e0 80087000 00000000 ntoskrnl!IopInitializeSystemDrivers+0x5b
f6413b7c 804f4820 80087000 00000001 00000000 ntoskrnl!IoInitSystem+0x3fe
kd> !cmreslist e11d8808

CmResourceList at 0xe11d8808  Version 0.0  Interface 0x1  Bus #0
  Entry 0 - Interrupt (0x2) Device Exclusive (0x1)
    Flags (0x01) - LATCHED
    Level 0x9, Vector 0x9, Affinity 0xffffffff
  Entry 1 - Port (0x1) Device Exclusive (0x1)
    Flags (0x01) - PORT_MEMORY PORT_IO
    Range starts at 0xdfe0 for 0x20 bytes (started with IRQ 9, IO dfe0-dfff)
  Entry 2 - DevicePrivate (0x81) Device Exclusive (0x1)
    Flags (0000) -
    Data - {0x00010120, 0000000000, 0000000000}

kd> g
```

Debugging a Service Application

12/5/2018 • 2 minutes to read • Edit Online

A *service*, also known as a *Windows service*, is a user-mode process designed to be started by Windows without human interaction. It is started automatically at system boot, or by an application that uses the service functions included in the Win32 API. A service can also be started by a human user through the Services control panel utility. Every service must conform to the interface rules of the service control manager (SCM).

Each service is composed of three elements: a *service application*, a *service control program*, and the service control manager itself. Although a service application is sometimes (incorrectly) referred to as a "service," it is actually one of the three components that make up a service. The service application can contain almost any kind of user-mode code. The service control program controls when the service application starts and stops. The service control manager is part of Windows.

The following sections describe how to debug a service application:

Choosing the Best Method

Preparing to Debug the Service Application

Debugging the Service Application Automatically

Debugging the Service Application Manually

For an overview of services, service applications, and the service control manager, see *Microsoft Windows Internals: Microsoft Windows Server 2003, Windows XP, and Windows 2000* by David A. Solomon and Mark E. Russinovich (4th edition, Microsoft Press, 2005).

Choosing the Best Method

12/5/2018 • 2 minutes to read • Edit Online

There are several different ways to debug a service application. In order to choose the correct method, you must first make two choices: the time at which the debugger is attached to the service application and what debugging configuration to use.

There are three stages at which the debugger can be attached to the service application:

- The very beginning of the service startup. The debugger is automatically launched when the service begins. Choose this option if you want to debug the service's initialization code.
- The first time that the service encounters an exception. The debugger is automatically launched when an exception or crash occurs or if the service application calls the **DebugBreak** function. Choose this option if you want the debugger to appear when a problem is encountered but not before.
- After the service is running normally. You can manually attach a debugger to a service that is already running at any time. Choose this option if you do not want to make advance preparations for debugging.

There are three debugging configurations you can choose:

- Local debugging. A single debugger, running on the same computer as the service.
- Remote debugging. A debugging server running on the same computer as the service, being controlled from a debugging client running on a second computer.
- Kernel-controlled user-mode debugging. A user-mode debugger running on the same computer as the service, being controlled from a kernel debugger on a second computer.

If your service is running on Windows 2000, Windows XP, or Windows Server 2003, you can combine any of these three attach options with any of these three debugging configuration options.

If your service is running on Windows Vista or a later version of Windows, there is one restriction on how these choices can be combined. If you want to debug from the beginning of the service startup, or from the time that an exception is encountered, you must use either remote debugging or kernel-controlled user-mode debugging.

In other words, on Windows Vista and later, you cannot use local debugging unless you plan to attach the debugger manually after the service is already running. This restriction results from the fact that in these versions of Windows, services run in session 0, and any debugger that is automatically launched and attached to the service is also in session 0, and does not have a user interface on the computer that the service is running on.

Preparing to Debug the Service Application

12/5/2018 • 11 minutes to read • Edit Online

This topic lists all the preparatory steps that may be required prior to debugging a service application. Which steps are required in your scenario depends on which attach option you have chosen and which debugging configuration you have chosen. For a list of these choices, see Choosing the Best Method.

Each of the preparatory steps described in this topic specifies the conditions under which it is required. These steps can be done in any order.

Enabling the Debugging of the Initialization Code

If you plan to debug the service application from the beginning of its execution, including its initialization code, this preparatory step is required.

Locate or create the following registry key, where *ProgramName* is the name of the service application's executable file:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\ProgramName
```

ProgramName should include the file name extension, but not the path. For example, *ProgramName* might be Myservice.exe or Thisservice.dll.

Under this registry key, create a string data value entitled **Debugger**. The value of this string should be set to the full path and file name of a debugger to be attached to the service application.

- If you plan to debug locally, use a string such as the following:

  ```
  c:\Debuggers\windbg.exe
  ```

  Do not choose this option if you are running Windows Vista or a later version of Windows.

- If you plan to use remote debugging, specify NTSD with the -noio option. This causes NTSD to run without any console of its own, accessible only through the remote connection. For example:

  ```
  c:\Debuggers\ntsd.exe -server ServerTransport -noio -y SymbolPath
  ```

  If your debugging session begins before Windows is fully loaded, you may not be able to access symbols from a remote share; in such a case, you must use local symbols. *ServerTransport* must specify a transport protocol that is implemented by the Windows kernel without interfacing with a user-mode service, such as TCP or NPIPE. For the syntax of *ServerTransport*, see **Activating a Debugging Server**.

- If you plan to control the user-mode debugger from a kernel-mode debugger, specify NTSD with the -d option. For example:

  ```
  c:\Debuggers\ntsd.exe -d -y SymbolPath
  ```

  If you plan to use this method and your user-mode symbols will be accessed from a symbol server, you should combine this method with remote debugging. In this case, specify NTSD with the -ddefer option. Choose a transport protocol that is implemented by the Windows kernel without interfacing with a user-mode service, such as TCP or NPIPE. For example:

```
c:\Debuggers\ntsd.exe -server ServerTransport -ddefer -y SymbolPath
```

For details, see Controlling the User-Mode Debugger from the Kernel Debugger.

After this registry edit is complete, the debugger is launched whenever a service with this name is started or restarted.

Enabling the Service Application to Break Into the Debugger

If you want the service application to break into the debugger when it crashes or encounters an exception, this preparatory step is required. This step is also required if you want the service application to break into the debugger by calling the **DebugBreak** function.

Note If you have enabled debugging of the initialization code (the step described in the subsection "Enabling the Debugging of the Initialization Code"), you should skip this step. When initialization code debugging is enabled, the debugger attaches to the service application when it starts, which causes all crashes, exceptions, and calls to **DebugBreak** to be routed to the debugger without additional preparations being needed.

This preparatory step involves registering the chosen debugger as the postmortem debugger. This is done by using the -iae or -iaec options on the debugger command line. We recommend the following commands, but if you want to vary them, see the syntax details in Enabling Postmortem Debugging.

- If you plan to debug locally, use a command such as the following:

  ```
  windbg -iae
  ```

  Do not choose this option if you are running Windows Vista or a later version of Windows.

- If you plan to use remote debugging, specify NTSD with the -noio option. This causes NTSD to run without any console of its own, accessible only through the remote connection. To install a postmortem debugger that includes the -server parameter, you must manually edit the registry; for details, see Enabling Postmortem Debugging. For example, the **Debugger** value of the **AeDebug** key could be the following:

  ```
  ntsd -server npipe:pipe=myproc%x -noio -p %ld -e %ld -g -y SymbolPath
  ```

  In the pipe specification, the **%x** token is replaced with the process ID of the process that launches the debugger. This guarantees that if more than one process launches a postmortem debugger, each has a unique pipe name. If your debugging session begins before Windows is fully loaded, you may not be able to access symbols from a remote share; in such a case, you must use local symbols. *ServerTransport* must specify a transport protocol that is implemented by the Windows kernel without interfacing with a user-mode service, such as TCP or NPIPE. For the syntax of *ServerTransport*, see **Activating a Debugging Server**.

- If you plan to control the user-mode debugger from a kernel-mode debugger, specify NTSD with the -d option. For example:

  ```
  ntsd -iaec -d -y SymbolPath
  ```

  If you choose this method and intend to access user-mode symbols from a symbol server, you should combine this method with remote debugging. In this case, specify NTSD with the -ddefer option. Choose a transport protocol that is implemented by the Windows kernel without interfacing with a user-mode service, such as TCP or NPIPE. To install a postmortem debugger that includes the -server parameter, you must manually edit the registry; for details, see Enabling Postmortem Debugging. For example, the **Debugger** value of the **AeDebug** key could be the following:

```
ntsd -server npipe:pipe=myproc%x -ddefer -p %ld -e %ld -g -y SymbolPath
```

For details, see Controlling the User-Mode Debugger from the Kernel Debugger.

When you issue one of these commands, the postmortem debugger is registered. This debugger will be launched whenever any user-mode program, including a service application, encounters an exception or runs a **DebugBreak** function.

Adjusting the Service Application Timeout

If you plan to launch the debugger automatically (either when the service starts or when it encounters an exception), this preparatory step is required.

Locate the following registry key:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control
```

Under this key, locate or create a DWORD data value called **ServicesPipeTimeout**. Set this entry to the amount of time in milliseconds that you want the service to wait before timing out. For example, a value of 60,000 is one minute, while a value of 86,400,000 is 24 hours. When this registry value is not set, the default timeout is about thirty seconds.

The significance of this value is that a clock starts to run when each service is launched, and when the timeout value is reached, any debugger attached to the service is terminated. Therefore, the value you choose should be longer than the total amount of time that elapses between the launching of the service and the completion of your debugging session.

This setting applies to every service that is started or restarted after the registry edit is complete. If some service crashes or hangs and this setting is still in effect, the problem is not detected by Windows. Therefore, you should use this setting only while you are debugging, and return the registry key to its original value after your debugging is complete.

Isolating the Service

Sometimes, multiple services are combined in a single Service Host (Svchost) process. If you want to debug such a service, you must first isolate it into a separate Svchost process.

There are three methods by which you can isolate a service. Microsoft recommends the Moving the Service to its Own Group method, as follows. The alternative methods (Changing the Service Type and Duplicating the SvcHost Binary) can be used on a temporary basis for debugging, but because they alter the way the service runs, they are not as reliable as the first method.

Preferred Method: Moving the Service to its Own Group

1. Issue the following Service Configuration tool (Sc.exe) command, where *ServiceName* is the name of the service:

   ```
   sc qc ServiceName
   ```

   This displays the current configuration values for the service. The value of interest is BINARY_PATH_NAME, which specifies the command line used to launch the service control program. In this scenario, because your service is not yet isolated, this command line includes a directory path, Svchost.exe, and some SvcHost parameters, including the -k switch, followed by a group name. For example, it may look something like this:

   ```
   %SystemRoot%\System32\svchost.exe -k LocalServiceNoNetwork
   ```

Remember this path and the group name; they are used in steps 5 and 6.

2. Locate the following registry key:

   ```
   HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\WindowsNT\CurrentVersion\SvcHost
   ```

   Create a new REG_MULTI_SZ value with a unique name (for example, **TempGrp**).

3. Set this new value equal to the name of the service that you want to isolate. Do not include any directory path or file name extension. For example, you might set the new value **TempGrp** equal to **MyService**.

4. Under the same registry key, create a new key with the same name you used in step 2. For example:

   ```
   HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\WindowsNT\CurrentVersion\SvcHost\TempGrp
   ```

   Now the SvcHost key contains a value with the new name and also has a subordinate key with this same name.

5. Look for another key subordinate to the SvcHost key that has the same name as the group you found in step 1. If such a key exists, examine all the values in it, and create duplicates of them in the new key you created in step 4.

   For example, the old key might be named this:

   ```
   HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\WindowsNT\CurrentVersion\SvcHost\LocalServiceNoNetwork
   ```

   and it might contain values such as **CoInitializeSecurityParam**, **AuthenticationCapabilities**, and other values. You would go to the newly created key:

   ```
   HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\WindowsNT\CurrentVersion\SvcHost\TempGrp
   ```

   and create values in it that are identical in name, type, and data to those in the old key.

   If the old key does not exist, you do not need to create a new key.

6. Use the following Service Configuration tool command to revise the path found in step 1:

   ```
   sc config ServiceName binPath= "RevisedPath"
   ```

   In this command, *ServiceName* is the name of the service, and *RevisedPath* is the new value you are supplying for BINARY_PATH_NAME. For *RevisedPath*, use the exact same path as the one displayed in step 1, including all the options shown on that line, making only one change: replace the parameter following the -k switch with the name of the new registry value you created in step 2. Enclose *RevisedPath* in quotation marks. The space after the equal sign is required.

   For example, your command might look like this:

   ```
   sc config MyService binPath= "%SystemRoot%\System32\svchost.exe -k TempGrp"
   ```

   You may want to use the **sc qc** command again to review the change you have made.

These settings will take effect the next time the service is started. To clear the effects of the old service, we recommend that you restart Windows rather than just restarting the service.

After you have completed your debugging, if you want to return this service to the shared service host, use the **sc config** command again to return the binary path to its original value, and delete the new registry keys and values you created..

Alternative Method: Changing the Service Type

1. Issue the following Service Configuration tool (Sc.exe) command, where *ServiceName* is the name of the service:

   ```
   sc config ServiceName type= own
   ```

   The space after the equal sign is required.

2. Restart the service, by using the following commands:

   ```
   net stop ServiceName
   net start ServiceName
   ```

This alternative is not the recommended method because it can alter the behavior of the service. If you do use this method, use the following command to revert to the normal behavior after you have completed your debugging:

```
sc config ServiceName type= share
```

Alternative Method: Duplicating the SvcHost Binary

1. The Svchost.exe executable file is located in the system32 directory of Windows. Make a copy of this file, name it svhost2.exe, and place it in the system32 directory as well.

2. Issue the following Service Configuration tool (Sc.exe) command, where *ServiceName* is the name of the service:

   ```
   sc qc ServiceName
   ```

   This command displays the current configuration values for the service. The value of interest is BINARY_PATH_NAME, which specifies the command line used to launch the service control program. In this scenario, because your service is not yet isolated, this command line will include a directory path, Svchost.exe, and probably some SvcHost parameters. For example, it may look something like this:

   ```
   %SystemRoot%\System32\svchost.exe -k LocalServiceNoNetwork
   ```

3. To revise this path, issue the following command:

   ```
   sc config ServiceName binPath= "RevisedPath"
   ```

   In this command, *ServiceName* is the name of the service, and *RevisedPath* is the new value you are supplying for BINARY_PATH_NAME. For *RevisedPath*, use the exact same path as the one displayed in step 2, including all the options shown on that line, making only one change: replace Svchost.exe with Svchost2.exe. Enclose *RevisedPath* in quotation marks. The space after the equal sign is required.

   For example, your command might look like this:

```
sc config MyService binPath= "%SystemRoot%\System32\svchost2.exe -k LocalServiceNoNetwork"
```

You can use the **sc qc** command again to review the change you have made.

4. Restart the service by using the following commands:

```
net stop ServiceName
net start ServiceName
```

This alternative is not the recommended method because it can alter the behavior of the service. If you do use this method, use the **sc config** command to change the path back to its original value after you have completed your debugging.

Debugging the Service Application Automatically

12/5/2018 • 2 minutes to read • Edit Online

A debugger can be launched automatically when the service application starts up. Alternatively, it can be launched automatically when the service application encounters an exception or executes a **DebugBreak** command. If you have chosen one of these methods, this topic explains how to proceed. If you are not sure which method to choose, see Choosing the Best Method.

Then use the following procedure:

1. Do one of the following preparatory steps:
   - If you plan to debug the service application from the very beginning, including its initialization code, follow the procedure described in Enabling the Debugging of the Initialization Code. Alternatively, if you want the service application to break into the debugger when it crashes or encounters an exception, follow the procedure described in Enabling the Service Application to Break Into the Debugger.
   - To assure that the service application will allow the debugger to run properly, perform the procedure described in Adjusting the Service Application Timeout.
   - If the service is combined with other services in a single SvcHost process, perform the procedure described in Isolating the Service.
2. If the service is already running, you must restart it for these changes to take effect. We recommend that you restart Windows itself, in order to remove any effects of the running service. If you do not want to restart Windows, use the following commands, where *ServiceName* is the name of the service:

   ```
   net stop ServiceName
   net start ServiceName
   ```

3. If you have chosen to debug the service application's initialization code, when the service starts, the debugger is launched and attaches to the service application.

   If you have chosen to let the debugger be triggered by an exception, the service application executes normally until it encounters an exception or executes a **DebugBreak** function. At this point, the debugger is launched and attaches to the service application.

4. The next step depends on the debugger command line you specified during step 1:
   - If you specified a debugger without any remoting options, this debugger is launched and its window becomes visible.
   - If you specified NTSD with the -server and -noio options, NTSD is launched without a console window. You can then connect to the debugging session from another computer by starting any user-mode debugger with the -remote parameter. For instructions, see **Activating a Debugging Client**.
   - If you specified NTSD with the -d option, NTSD is launched without a console window. You can then connect to the debugging session by using kernel debugger running on another computer. For instructions, see Controlling the User-Mode Debugger from the Kernel Debugger.
   - If you specified NTSD with the -ddefer and -server options, NTSD is launched without a console window. You can then connect to the debugging session by using both a kernel debugger and a user-mode remote debugger, running on a different computer than the service (but possibly the same computer as each other). For instructions, see Combining This Method with Remote Debugging.
5. When the debugger starts, the service pauses at the initial process breakpoint, the exception, or the **DebugBreak** command. This enables you to examine the current state of the service application, set

breakpoints, and make any other desired configuration choices.

6. Use **g (Go)** or another execution command to resume the execution of the service application.

Related topics

DebugBreak function

Debugging the Service Application Manually

12/5/2018 • 2 minutes to read • Edit Online

Manually attaching to a service application after it has been started is much like debugging any running user-mode process.

Use the TList tool with the **/s** option to display the process ID (PID) of each running process and the services active in each process.

If the service application you want to debug is combined with other services in a single process, you must isolate it before debugging it. To do this, perform the procedure described in Isolating the Service. At the end of this procedure, restart the service.

To determine the new PID of the service, issue the following Service Configuration tool (Sc.exe) command, where *ServiceName* is the name of the service:

```
sc queryex ServiceName
```

Now start WinDbg or CDB with this service application as the target. There are three ways to do this: by specifying the PID with the -p option, by specifying the executable name with the -pn option (if the executable name is unique), or by specifying the service name with the -psn option.

For example, if the process SpoolSv.exe has a PID of 651 and contains the service named *Spooler*, the following three commands are equivalent:

```
windbg -p 651 [AdditionalOptions]
windbg -pn spoolsv.exe [AdditionalOptions]
windbg -psn spooler [AdditionalOptions]
```

After the debugger starts, proceed as you would in any other user-mode debugging session.

Symbols for Windows debugging (WinDbg, KD, CDB, NTSD)

10/7/2019 • 2 minutes to read • Edit Online

Symbols for the Windows debuggers (WinDbg, KD, CDB, and NTSD) are available from a public symbol server.

This section includes:

Introduction to Symbols

Accessing Symbols for Debugging

How the Debugger Recognizes Symbols

Symbol Problems While Debugging

These topics explain what symbols are, how to access them during a debugging session, how to control the debugger's symbol options and symbol matching, and how to respond to various symbol-related problems during debugging.

If you simply want to configure your debugger to access symbols for your own programs and for Windows, you may find it quicker to read the less-detailed introductory topics Symbol Path and Using a Symbol Server.

Introduction to Symbols

12/5/2018 • 2 minutes to read • Edit Online

This section includes:

Symbols and Symbol Files

Public and Private Symbols

Symbol path for Windows debuggers

10/24/2019 • 6 minutes to read • Edit Online

The symbol path specifies locations where the Windows debuggers (WinDbg, KD, CDB, NTST) look for symbol files. For more information about symbols and symbol files, see Symbols.

Some compilers (such as Microsoft Visual Studio) put symbol files in the same directory as the binary files. The symbol files and the checked binary files contain path and file name information. This information frequently enables the debugger to find the symbol files automatically. If you are debugging a user-mode process on the computer where the executable was built, and if the symbol files are still in their original location, the debugger can locate the symbol files without you setting the symbol path.

In most other situations, you have to set the symbol path to point to your symbol file locations.

Symbol path syntax

The debugger's symbol path is a string that consists of multiple directory paths, separated by semicolons.

Relative paths are supported. However, unless you always start the debugger from the same directory, you should add a drive letter or a network share before each path. Network shares are also supported.

For each directory in the symbol path, the debugger looks in three directories. For example, if the symbol path includes the `c:\MyDir` directory, and the debugger is looking for symbol information for a DLL, the debugger first looks in `c:\MyDir\symbols\dll`, then in `c:\MyDir\dll`, and finally in `c:\MyDir`. The debugger then repeats this process for each directory in the symbol path. Finally, the debugger looks in the current directory and then in the current directory with `..\dll` appended to it. (The debugger appends `..\dll`, `..\exe`, or `..\sys`, depending on which binaries it is debugging.)

Symbol files have date and time stamps. You do not have to worry that the debugger will use the wrong symbols that it may find first in this sequence. It always looks for the symbols that match the time stamp on the binary files that it is debugging. For more information about responses when symbols files are not available, see Compensating for Symbol-Matching Problems.

One way to set the symbol path is by entering the **.sympath** command. For other ways to set the symbol path, see Controlling the Symbol Path later in this topic.

Caching symbols locally

We strongly recommend that you always cache your symbols locally. One way to cache symbols locally is to include `cache*;` or `cache*localsymbolcache;*` in your symbol path.

If you include the string `cache*;` in your symbol path, symbols loaded from any element that appears to the right of this string are stored in the default symbol cache directory on the local computer. For example, the following command tells the debugger to get symbols from the network share `\\someshare` and cache the symbols in the default location on the local computer.

```
.sympath cache*;\\someshare
```

If you include the string `cache*localsymbolcache;` in your symbol path, symbols loaded from any element that appears to the right of this string are stored in the *localsymbolcache* directory.

For example, the following command tells the debugger to obtain symbols from the network share `\\someshare`

and cache the symbols in the `c:\MySymbols` directory.

```
.sympath cache*c:\MySymbols;\\someshare
```

Using a symbol server

If you are connected to the Internet or a corporate network, the most efficient way to access symbols is to use a symbol server. You can use a symbol server by using the `srv*`, `srv*symbolstore`, or `srv*localsymbolcache*symbolstore` string in your symbol path.

If you include the string `srv*` in your symbol path, the debugger uses a symbol server to get symbols from the default symbol store. For example, the following command tells the debugger to use a symbol server to get symbols from the default symbol store. These symbols are not cached on the local computer.

```
.sympath srv*
```

If you include the string `srv*symbolstore` in your symbol path, the debugger uses a symbol server to get symbols from the *symbolstore* store. For example, the following command tells the debugger to use a symbol server to get symbols from the symbol store at https://msdl.microsoft.com/download/symbols. These symbols are not cached on the local computer.

```
.sympath srv*https://msdl.microsoft.com/download/symbols
```

If you include the string `srv*localcache*symbolstore` in your symbol path, the debugger uses a symbol server to get symbols from the *symbolstore* store and caches them in the *localcache* directory. For example, the following command tells the debugger to use a symbol server to get symbols from the symbol store at https://msdl.microsoft.com/download/symbols and cache the symbols in `c:\MyServerSymbols`.

```
.sympath srv*c:\MyServerSymbols*https://msdl.microsoft.com/download/symbols
```

If you have a directory on your computer where you manually place symbols, do not use that directory as the cache for symbols obtained from a symbol server. Instead, use two separate directories. For example, you can manually place symbols in `c:\MyRegularSymbols` and then designate `c:\MyServerSymbols` as a cache for symbols obtained from a server. The following example shows how to specify both directories in your symbol path.

```
.sympath c:\MyRegularSymbols;srv*c:\MyServerSymbols*https://msdl.microsoft.com/download/symbols
```

For more information about symbol servers, see Symbol Stores and Symbol Servers.

Combining cache* and srv*

If you include the string `cache*;` in your symbol path, symbols loaded from any element that appears to the right of this string are stored in the default symbol cache directory on the local computer. For example, the following command tells the debugger to use a symbol server to get symbols from the store at https://msdl.microsoft.com/download/symbols and cache them in the default symbol cache directory.

```
.sympath cache*;srv*https://msdl.microsoft.com/download/symbols
```

If you include the string `cache*localsymbolcache;` in your symbol path, symbols loaded from any element that

appears to the right of this string are stored in the *localsymbolcache* directory.

For example, the following command tells the debugger to use a symbol server to get symbols from the store at https://msdl.microsoft.com/download/symbols and cache the symbols in the `c:\MySymbols` directory.

```
.sympath cache*c:\MySymbols;srv*https://msdl.microsoft.com/download/symbols
```

Using AgeStore to reduce the cache size

You can use the AgeStore tool to delete cached files that are older than a specified date, or to delete enough old files that the resulting size of the cache is less than a specified amount. This can be useful if your downstream store is too large. For details, see AgeStore.

For more information about symbol servers and symbol stores, see Symbol Stores and Symbol Servers.

Lazy symbol loading

The debugger's default behavior is to use *lazy symbol loading* (also known as *deferred symbol loading*). This kind of loading means that symbols are not loaded until they are required.

When the symbol path is changed, for example by using the `.sympath` command, all loaded modules with export symbols are lazily reloaded.

Symbols of modules with full PDB symbols will be lazily reloaded if the new path no longer includes the original path that was used to load the PDB symbols. If the new path still includes the original path to the PDB symbol file, those symbols will not be lazily reloaded.

For more information about lazy symbol loading, see Deferred Symbol Loading.

You can turn off lazy symbol loading in CDB and KD by using the `-s` command-line option. You can also force symbol loading by using the `ld` **(Load Symbols)** command or by using the `.reload` **(Reload Module)** command together with the `/f` option.

Azure DevOps Services Artifacts

A symbol server is available with Azure Artifacts in Azure DevOps Services. For information on working with Azure Artifacts in WinDbg, see Debug with symbols in WinDbg. For general information about Azure generated symbols, see Symbol files (PDBs).

Controlling the symbol path

To control the symbol path, you can do one of the following:

- Use the `.sympath` command to display, set, change, or append to the path. The `.symfix` **(Set Symbol Store Path)** command is similar to `.sympath` but saves you some typing.
- Before you start the debugger, use the `_NT_SYMBOL_PATH` and `_NT_ALT_SYMBOL_PATH` environment variables to set the path. The symbol path is created by appending `_NT_SYMBOL_PATH` after `_NT_ALT_SYMBOL_PATH`. (Typically, the path is set through the `_NT_SYMBOL_PATH`. However, you might want to use `_NT_ALT_SYMBOL_PATH` to override these settings in special cases, such as if you have private versions of shared symbol files.) If you try to add an invalid directory through these environment variables, the debugger ignores this directory.
- When you start the debugger, use the `-y` command-line option to set the path.
- (WinDbg only) Use the File | Symbol File Path command or press `CTRL+S` to display, set, change, or append to the path.

If you use the `-sins` command-line option, the debugger ignores the symbol path environment variable.

Related topics

Advanced SymSrv Use

Symbols and Symbol Files

1/11/2019 • 2 minutes to read • Edit Online

When applications, libraries, drivers, or operating systems are linked, the linker that creates the .exe and .dll files also creates a number of additional files known as *symbol files*.

Symbol files hold a variety of data which are not actually needed when running the binaries, but which could be very useful in the debugging process.

Typically, symbol files might contain:

- Global variables
- Local variables
- Function names and the addresses of their entry points
- Frame pointer omission (FPO) records
- Source-line numbers

Each of these items is called, individually, a *symbol*. For example, a single symbol file Myprogram.pdb might contain several hundred symbols, including global variables and function names and hundreds of local variables. Often, software companies release two versions of each symbol file: a full symbol file containing both *public symbols* and *private symbols*, and a reduced (stripped) file containing only public symbols. For details, see Public and Private Symbols.

When debugging, you must make sure that the debugger can access the symbol files that are associated with the target you are debugging. Both live debugging and debugging crash dump files require symbols. You must obtain the proper symbols for the code that you wish to debug, and load these symbols into the debugger.

Windows Symbols

Windows keeps its symbols in files with the extension .pdb.

The compiler and the linker control the symbol format. The Visual C++ linker, places all symbols into .pdb files.

The Windows operating system is built in two versions. The *free build* (or *retail build*) has relatively small binaries, and the *checked build* (or *debug build*) has larger binaries, with more debugging symbols in the code itself. Each of these builds has its own symbol files. When debugging a target on Windows, you must use the symbol files that match the build of Windows on the target.

The following table lists several of the directories which exist in a standard Windows symbol tree:

DIRECTORY	CONTAINS SYMBOL FILES FOR
ACM	Microsoft Audio Compression Manager files
COM	Executable files (.com)
CPL	Control Panel programs

DIRECTORY	CONTAINS SYMBOL FILES FOR
DLL	Dynamic-link library files (.dll)
DRV	Driver files (.drv)
EXE	Executable files (.exe)
SCR	Screen-saver files
SYS	Driver files (.sys)

Public and Private Symbols

12/5/2018 • 8 minutes to read • Edit Online

When a full-sized .pdb or .dbg symbol file is built by a linker, it contains two distinct collections of information: the *private symbol data* and a *public symbol table*. These collections differ in the list of items they contain and the information they store about each item.

The private symbol data includes the following items:

- Functions
- Global variables
- Local variables
- Information about user-defined structures, classes, and data types
- The name of the source file and the line number in that file corresponding to each binary instruction

The public symbol table contains fewer items:

- Functions (except for functions declared **static**)
- Global variables specified as **extern** (and any other global variables visible across multiple object files)

As a general rule, the public symbol table contains exactly those items that are accessible from one source file to another. Items visible in only one object file--such as **static** functions, variables that are global only within a single source file, and local variables--are not included in the public symbol table.

These two collections of data also differ in what information they include for each item. The following information is typically included for each item contained in the private symbol data:

- Name of the item
- Address of the item in virtual memory
- Frame pointer omission (FPO) records for each function
- Data type of each variable, structure, and function
- Types and names of the parameters for each function
- Scope of each local variable
- Symbols associated with each line in each source file

On the other hand, the public symbol table stores only the following information about each item included in it:

- The name of the item.
- The address of the item in the virtual memory space of its module. For a function, this is the address of its entry point.
- Frame pointer omission (FPO) records for each function.

In other words, the public symbol data can be thought of as a subset of the private symbol data in two ways: it contains a shorter list of items, and it also contains less information about each item. For example, the public symbol data does not include local variables at all. Each local variable is included only in the private symbol data,

with its address, data type, and scope. Functions, on the other hand, are included both in the private symbol data and public symbol table, but while the private symbol data includes the function name, address, FPO records, input parameter names and types, and output type, the public symbol table includes just the function name, address, and FPO record.

There is one other difference between the private symbol data and the public symbol table. Many of the items in the public symbol table have names that are *decorated* with a prefix, a suffix, or both. These decorations are added by the C compiler, the C++ compiler, and the MASM assembler. Typical prefixes include a series of underscores or the string **__imp_** (designating an imported function). Typical suffixes include one or more at signs (**@**) followed by addresses or other identifying strings. These decorations are used by the linker to disambiguate the symbol, since it is possible that function names or global variable names could be repeated across different modules. These decorations are an exception to the general rule that the public symbol table is a subset of the private symbol data.

Full Symbol Files and Stripped Symbol Files

A *full symbol file* contains both the private symbol data and the public symbol table. This kind of file is sometimes referred to as a *private symbol file*, but this name is misleading, for such a file contains both private and public symbols.

A *stripped symbol file* is a smaller file that contains only the public symbol table - or, in some cases, only a subset of the public symbol table. This file is sometimes referred to as a *public symbol file*.

Creating Full and Stripped Symbol Files

If you build your binaries with Visual Studio, you can create either full or stripped symbol files. When building a "debug build" of a binary, Visual Studio typically will create full symbol files. When building a "retail build", Visual Studio typically creates no symbol files, but a full or stripped symbol file will be created if the proper options are set.

If you build your binaries with the Build utility, the utility will create full symbol files.

Using the BinPlace tool, you can create a stripped symbol file from a full symbol file. When the most common BinPlace options are used (**-a -x -s -n**), the stripped symbol files are placed in the directory that is listed after the **-s** switch, and the full symbol files are placed in the directory that is listed after the **-n** switch. When BinPlace strips a symbol file, the stripped and full versions of the file are given identical signatures and other identifying information. This allows you to use either version for debugging.

Using the PDBCopy tool, you can create a stripped symbol file from a full symbol file by removing the private symbol data. PDBCopy can also remove a specified subset of the public symbol table. For details, see PDBCopy.

Using the SymChk tool, you can determine whether a symbol file contains private symbols. For details, see SymChk.

Viewing Public and Private Symbols in the Debugger

You can use WinDbg, KD, or CDB to view symbols. When one of these debuggers has access to a full symbol file, it has both the information listed in the private symbol data and the information listed in the public symbol table. The private symbol data is more detailed, while the public symbol data contains symbol decorations.

When accessing private symbols, private symbol data is always used because these symbols are not included in the public symbol table. These symbols are never decorated.

When accessing public symbols, the debugger's behavior depends on certain symbol options:

- When the SYMOPT_UNDNAME option is on, decorations are not included when the name of a public symbol is displayed. Moreover, when searching for symbols, decorations are ignored. When this option is off, decorations are displayed when displaying public symbols, and decorations are used in searches. Private symbols are never decorated in any circumstances. This option is on by default in all debuggers.

- When the SYMOPT_PUBLICS_ONLY option is on, private symbol data is ignored, and only the public symbol table is used. This option is off by default in all debuggers.
- When the SYMOPT_NO_PUBLICS option is on, the public symbol table is ignored, and searches and symbol information use the private symbol data alone. This option is off by default in all debuggers.
- When the SYMOPT_AUTO_PUBLICS option is on (and both SYMOPT_PUBLICS_ONLY and SYMOPT_NO_PUBLICS are off), the first symbol search is performed in the private symbol data. If the desired symbol is found there, the search terminates. If not, the public symbol table is searched. Since the public symbol table contains a subset of the symbols in the private data, normally this results in the public symbol table being ignored.
- When the SYMOPT_PUBLICS_ONLY, SYMOPT_NO_PUBLICS, and SYMOPT_AUTO_PUBLICS options are all off, both private symbol data and the public symbol table are searched each time a symbol is needed. However, when matches are found in both places, the match in the private symbol data is used. Therefore, the behavior in this instance is the same as when SYMOPT_AUTO_PUBLICS is on, except that using SYMOPT_AUTO_PUBLICS may cause symbol searches to happen slightly faster.

Here is an example in which the command **x (Examine Symbols)** is used three times. The first time, the default symbol options are used, and so the information is taken from the private symbol data. Note that this includes information about the address, size, and data type of the array **typingString**. Next, the command .symopt+ 4000 is used, causing the debugger to ignore the private symbol data. When the **x** command is then run again, the public symbol table is used; this time there is no size and data type information for **typingString**. Finally, the command .symopt- 2 is used, which causes the debugger to include decorations. When the **x** command is run this final time, the decorated version of the function name, **_typingString**, is shown.

```
0:000> x /t /d *!*typingstring*
00434420 char [128] TimeTest!typingString = char [128] ""

0:000> .symopt+ 4000

0:000> x /t /d *!*typingstring*
00434420 <NoType> TimeTest!typingString = <no type information>

0:000> .symopt- 2

0:000> x /t /d *!*typingstring*
00434420 <NoType> TimeTest!_typingString = <no type information>
```

Viewing Public and Private Symbols with the DBH Tool

Another way to view symbols is by using the the DBH tool. DBH uses the same symbol options as the debugger. Like the debugger, DBH leaves SYMOPT_PUBLICS_ONLY and SYMOPT_NO_PUBLICS off by default, and turns SYMOPT_UNDNAME and SYMOPT_AUTO_PUBLICS on by default. These defaults can be overridden by a command-line option or by a DBH command.

Here is an example in which the DBH command **addr 414fe0** is used three times. The first time, the default symbol options are used, and so the information is taken from the private symbol data. Note that this includes information about the address, size, and data type of the function **fgets**. Next, the command symopt +4000 is used, which causes DBH to ignore the private symbol data. When the **addr 414fe0** is then run again, the public symbol table is used; this time there is no size and data type information for the function **fgets**. Finally, the command symopt -2 is used, which causes DBH to include decorations. When the **addr 414fe0** is run this final time, the decorated version of the function name, **_fgets**, is shown.

```
pid:4308 mod:TimeTest[400000]: addr 414fe0

fgets
   name : fgets
   addr :   414fe0
   size : 113
  flags : 0
   type : 7e
modbase :   400000
  value :        0
    reg : 0
  scope : SymTagNull (0)
    tag : SymTagFunction (5)
  index : 7d

pid:4308 mod:TimeTest[400000]: symopt +4000

Symbol Options: 0x10c13
Symbol Options: 0x14c13

pid:4308 mod:TimeTest[400000]: addr 414fe0

fgets
   name : fgets
   addr :   414fe0
   size : 0
  flags : 0
   type : 0
modbase :   400000
  value :        0
    reg : 0
  scope : SymTagNull (0)
    tag : SymTagPublicSymbol (a)
  index : 7f

pid:4308 mod:TimeTest[400000]: symopt -2

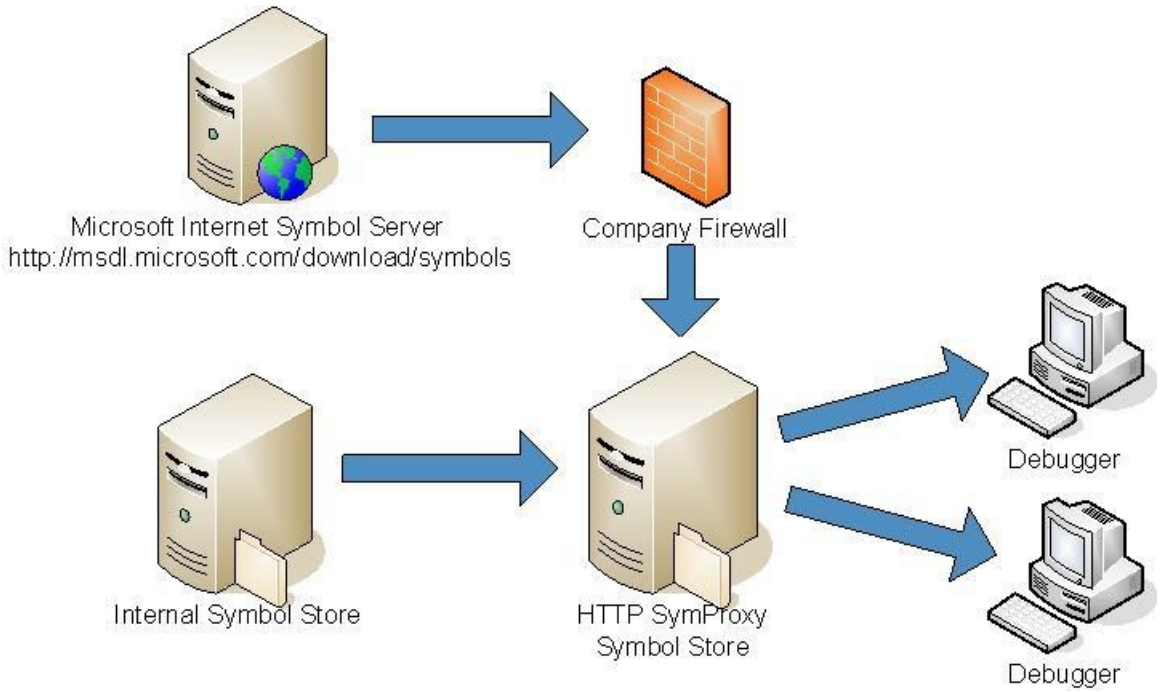
Symbol Options: 0x14c13
Symbol Options: 0x14c11

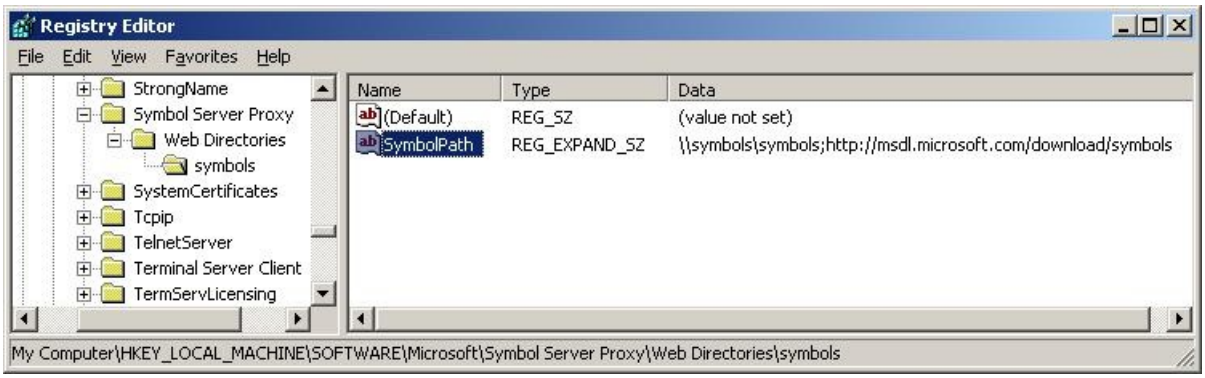
pid:4308 mod:TimeTest[400000]: addr 414fe0

_fgets
   name : _fgets
   addr :   414fe0
   size : 0
  flags : 0
   type : 0
modbase :   400000
  value :        0
    reg : 0
  scope : SymTagNull (0)
    tag : SymTagPublicSymbol (a)
  index : 7f
```

Accessing Symbols for Debugging

12/5/2018 • 2 minutes to read • Edit Online

Setting up symbols correctly for debugging can be a challenging task, particularly for kernel debugging. It often requires that you know the names and releases of all products on your computer. The debugger must be able to locate each of the symbol files corresponding to the product releases and service packs.

This can result in an extremely long symbol path consisting of a long list of directories.

To simplify these difficulties in coordinating symbol files, the symbol files can be gathered into a *symbol store*, which is then accessed by a *symbol server*.

A symbol store is a collection of symbol files, an index, and a tool that can be used by an administrator to add and delete files. The files are indexed according to unique parameters such as the time stamp and image size. A symbol store can also hold executable image files which can be extracted using a symbol server. Debugging Tools for Windows contains a symbol store creation tool called SymStore.

A symbol server enables the debuggers to automatically retrieve the correct symbol files from a symbol store without the user needing to know product names, releases, or build numbers. Debugging Tools for Windows contains a symbol server called SymSrv. The symbol server is activated by including a certain text string in the symbol path. Each time the debugger needs to load symbols for a newly loaded module, it calls the symbol server to locate the appropriate symbol files.

If you wish to use a different method for your symbol search than that provided by SymSrv, you can create your own symbol server DLL. For details on implementing such a symbol server, see Other Symbol Servers.

If you are performing user-mode debugging, you will need symbols for the target application. If you are performing kernel-mode debugging, you will need symbols for the driver you are debugging, as well as the Windows public symbols. Microsoft has created a symbol store with public symbols for Microsoft products; this symbol store is available on the internet. These symbols can be loaded using the **.symfix (Set Symbol Store Path)** command, as long as you have access to the internet while your debugger is running. For more information or to determine how to manually install these symbols, see Installing Windows Symbol Files.

This section includes:

Installing Windows Symbol Files

Symbol Stores and Symbol Servers

Deferred Symbol Loading

Installing Windows Symbol Files

12/5/2018 • 2 minutes to read • Edit Online

Before you debug the Windows kernel, a driver or app, you need access to the proper symbol files. The official way to get Windows symbols is to use the Microsoft Symbol Server. The symbol server makes symbols available to your debugging tools as needed. After a symbol file is downloaded from the symbol server it is cached on the local computer for quick access.

You can connect to the Microsoft Symbol Server with one simple use of the **.symfix (Set Symbol Store Path)** command. For full details, see Microsoft Public Symbols.

> **IMPORTANT**
>
> We are no longer publishing the offline symbol packages for Windows. The faster Windows update cadence means the Windows debugging symbols are quickly made out of date. We have made significant improvements to the online Microsoft Symbol Server where symbols for all Windows versions and updates are available. You can find more about this in this blog entry.
>
> For information on how to retrieve symbols for a machine that is not connected to the Internet, see Using a Manifest File with SymChk.

If you are going to debug a user-mode app, you need to install the symbols for this app as well.

You can debug an app if you have its symbols but not Windows symbols. However, your results will be much more limited. You will still be able to step through the app code, but any debugger activity which requires analysis of the kernel (such as getting a stack trace) is likely to fail.

Symbol Stores and Symbol Servers

4/9/2019 • 2 minutes to read • Edit Online

A *symbol store* is a collection of symbol files, an index, and a tool for adding and deleting files. A symbol store may also contain executable image files. The debugger accesses the files in a symbol store by using a *symbol server*. Debugging Tools for Windows includes both a symbol store creation tool, SymStore, and a symbol server, SymSrv. It also includes a tool, SymProxy, for setting up an HTTP symbol store on a network to serve as a proxy for all symbol stores that the debugger may need to access.

This section includes:

SymSrv

Using a Symbol Server

HTTP Symbol Stores

File Share (SMB) Symbol Server

SymStore

SymProxy

If you are not setting up your own symbol store, but just intend to use the public Microsoft symbol store, see Microsoft Public Symbols.

Using a Symbol Server

12/5/2018 • 2 minutes to read • Edit Online

A symbol server enables the debugger to automatically retrieve the correct symbol files from a symbol store - an indexed collection of symbol files - without the user needing to know product names, releases, or build numbers. The Debugging Tools for Windows package includes the symbol server SymSrv (symsrv.exe).

Using SymSrv with a Debugger

SymSrv can be used with WinDbg, KD, NTSD, or CDB.

To use this symbol server with the debugger, simply include the text **srv*** in the symbol path. For example:

```
set _NT_SYMBOL_PATH = srv*DownstreamStore*SymbolStoreLocation
```

where *DownstreamStore* specifies the local directory or network share that will be used to cache individual symbol files, and *SymbolStoreLocation* is the location of the symbol store either in the form *\\server\share* or as an internet address. For more syntax options, see Advanced SymSrv Use.

Microsoft has a Web site that makes Windows symbols publicly available. You can refer directly to this site in your symbol path in the following manner:

```
set _NT_SYMBOL_PATH=srv*DownstreamStore*https://msdl.microsoft.com/download/symbols
```

where, again, *DownstreamStore* specifies the local directory or network share that will be used to cache individual symbol files. For more information, see Microsoft Public Symbols.

If you plan to create a symbol store, configure a symbol store for web (HTTP) access, or write your own symbol server or symbol store, see Symbol Stores and Symbol Servers.

Using AgeStore to Reduce the Cache Size

Any symbol files downloaded by SymSrv will remain on your hard drive after the debugging session is over. To control the size of the symbol cache, the AgeStore tool can be used to delete cached files that are older than a specified date, or to reduce the contents of the cache below a specified size. For details, see AgeStore.

SymSrv

12/5/2018 • 2 minutes to read • Edit Online

SymSrv (symsrv.dll) is a symbol server that is included in the Debugging Tools for Windows package.

Many users of Debugging Tools for Windows use the public symbol store on the Microsoft Web site to access symbols for Microsoft products. If this is your goal, you only need to read the first topic listed below.

This section includes:

Microsoft Public Symbols

Advanced SymSrv Use

Firewalls and Proxy Servers

Microsoft public symbol server

11/1/2019 • 2 minutes to read • Edit Online

Server Status: No known issues □
The Microsoft public symbol server is fully operational.
Please report any known issues to windbgfb@microsoft.com.

The Microsoft symbol server makes Windows debugger symbols publicly available.

You can refer directly to the public symbol server in your symbol path in the following manner:

```
set _NT_SYMBOL_PATH=srv*DownstreamStore*https://msdl.microsoft.com/download/symbols
```

DownstreamStore must specify a directory on your local computer or network that will be used to cache symbols. This downstream store holds symbols that the debugger has accessed; the vast majority of symbols that have never been accessed remain on the symbol store at Microsoft. This keeps your downstream store relatively small and allows the symbol server to work quickly, only downloading each file once.

To avoid typing this long symbol path, use the **.symfix (Set Symbol Store Path)** command. The following command appends the public symbol store to your existing symbol path:

```
.symfix+ C:\MySymbols
```

If local symbol cache location is omitted, the sym subdirectory of the debugger installation directory will be used.

Use the **.sympath (Set Symbol Store Path)** command to display the full symbol path. This example shows how to use symfix to create a local symbol cache and use the Microsoft http symbol server.

```
0: kd> .symfix c:\MyCache
0: kd> .sympath
Symbol search path is: srv*
Expanded Symbol search path is: cache*c:\MyCache;SRV*https://msdl.microsoft.com/download/symbols
```

For more information about working with symbols, see the Symbol path for Windows debuggers.

Symbol File Compression

The Microsoft Symbol Server provides compressed versions of the symbol files. The files have an underscore at the end of the filename's extension to indicate that they are compressed. For example, the PDB for ntdll.dll is available as ntdll.pd_. When SymProxy downloads a compressed file, it will store the file decompressed in the local file system. The DontUncompress registry key can be set to disable this behavior in SymProxy.

Refer to the Debugger topic SymStore for information on using SymStore.exe /compress to store your own symbols compressed on your symbol server.

Advanced SymSrv Use

10/25/2019 • 10 minutes to read • Edit Online

SymSrv can deliver symbol files from a centralized symbol store. This store can contain any number of symbol files, corresponding to any number of programs or operating systems. The store can also contain binary files (this is useful when debugging minidumps).

The store can contain the actual symbol and binary files, or it can simply contain pointers to symbol files. If the store contains pointers, SymSrv will retrieve the actual files directly from their sources.

SymSrv can also be used to separate a large symbol store into a smaller subset that is appropriate for a specialized debugging task.

Finally, SymSrv can obtain symbol files from an HTTP or HTTPS source using the logon information provided by the operating system. SymSrv supports HTTPS sites protected by smartcards, certificates, and regular logins and passwords. For more information, see HTTP Symbol Stores.

Setting the Symbol Path

To use this symbol server, symsrv.dll must be installed in the same directory as the debugger. The symbol path can be set as shown here:

```
set _NT_SYMBOL_PATH = symsrv*ServerDLL*DownstreamStore*\\Server\Share

set _NT_SYMBOL_PATH = symsrv*ServerDLL*\\Server\Share

set _NT_SYMBOL_PATH = srv*DownstreamStore*\\Server\Share

set _NT_SYMBOL_PATH = srv*\\Server\Share
```

The parts of this syntax are explained as follows:

symsrv
This keyword must always appear first. It indicates to the debugger that this item is a symbol server, not just a normal symbol directory.

ServerDLL
Specifies the name of the symbol server DLL. If you are using the SymSrv symbol server, this will always be symsrv.dll.

srv
This is shorthand for **symsrv*symsrv.dll**.

DownstreamStore
Specifies the downstream store. This is a local directory or network share that will be used to cache individual symbol files.

You can specify more than one downstream store, separated by asterisks. Multiple downstream stores are explained in **Cascading Downstream Stores** further down on this page.

If you include two asterisks in a row where a downstream store would normally be specified, then the default downstream store is used. This store will be located in the sym subdirectory of the home directory. The home directory defaults to the debugger installation directory; this can be changed by using the **!homedir** extension or by setting the DBGHELP_HOMEDIR environment variable.

If *DownstreamStore* specifies a directory that does not exist, SymStore will attempt to create it.

If the *DownstreamStore* parameter is omitted and no extra asterisk is included -- in other words, if you use **srv** with exactly one asterisk or **symsrv** with exactly two asterisks -- then no downstream store will be created. The debugger will load all symbol files directly from the server, without caching them locally.

Note If you are accessing symbols from an HTTP or HTTPS site, or if the symbol store uses compressed files, a downstream store is always used. If no downstream store is specified, one will be created in the sym subdirectory of the home directory.

*Server**Share*
Specifies the server and share of the remote symbol store.

If a downstream store is used, the debugger will first look for a symbol file in this store. If the symbol file is not found, the debugger will locate the symbol file from the specified *Server* and *Share*, and then cache a copy of this file in the downstream store. The file will be copied to a subdirectory in the tree under *DownstreamStore* which corresponds to its location in the tree under *Server**Share*.

The symbol server does not have to be the only entry in the symbol path. If the symbol path consists of multiple entries, the debugger checks each entry for the needed symbol files, in order (from left to right), regardless of whether a symbol server or an actual directory is named.

Here are some examples. To use SymSrv as the symbol server with a symbol store on \\mybuilds\mysymbols, set the following symbol path:

```
set _NT_SYMBOL_PATH= symsrv*symsrv.dll*\\mybuilds\mysymbols
```

To set the symbol path so that the debugger will copy symbol files from a symbol store on \\mybuilds\mysymbols to your local directory c:\localsymbols, use:

```
set _NT_SYMBOL_PATH=symsrv*symsrv.dll*c:\localsymbols*\\mybuilds\mysymbols
```

To set the symbol path so that the debugger will copy symbol files from the HTTPS site `https://www.company.com/manysymbols` to a local network directory \\localserver\myshare\mycache, use:

```
set _NT_SYMBOL_PATH=symsrv*symsrv.dll*\\localserver\myshare\mycache*https://www.company.com/manysymbols
```

This last example can also be shortened as such:

```
set _NT_SYMBOL_PATH=srv*\\localserver\myshare\mycache*https://www.company.com/manysymbols
```

In addition, the symbol path can contain several directories or symbol servers, separated by semicolons. This allows you to locate symbols from multiple locations (or even multiple symbol servers). If a binary has a mismatched symbol file, the debugger cannot locate it using the symbol server because it checks only for the exact parameters. However, the debugger may find a mismatched symbol file with the correct name, using the traditional symbol path, and successfully load it. Even though the file is technically not the correct symbol file, it might provide useful information.

Compressed Files

SymSrv is compatible with symbol stores that contain compressed files, as long as this compression has been done with the compress.exe tool, which is available here. Compressed files should have an underscore as the last character in their file extensions (for example, module1.pd_ or module2.db_). For details, see SymStore.

If the files on the store are compressed, you must use a downstream store. SymSrv will uncompress all files before

caching them on the downstream store.

Deleting the Cache

If you are using a *DownstreamStore* as a cache, you can delete this directory at any time to save disk space.

It is possible to have a vast symbol store that includes symbol files for many different programs or Windows versions. If you upgrade the version of Windows used on your target computer, the cached symbol files will all match the earlier version. These cached files will not be of any further use, and therefore this might be a good time to delete the cache.

Cascading Downstream Stores

You can specify any number of downstream stores, separated by asterisks. These stores are known as *cascading symbol stores*.

After the initial **srv*** or **symsrv*ServerDLL******, each subsequent token represents a symbol location. The token furthest left is checked first. An empty token -- indicated by two asterisks in a row, or by an asterisk at the end of the string -- represents the default downstream store.

Here is an example of a symbol path that uses two downstream stores to hold information from the main symbol store being accessed. These could be called the master store, the mid-level store, and the local cache:

```
srv*c:\localcache*\\interim\store*https://msdl.microsoft.com/download/symbols
```

In this scenario, SymSrv will first look in c:\localcache for a symbol file. If it is found there, it will return a path to it. If it is not found there, it will look in \\interim\store. If the symbol file is found there, SymSrv will copy it to c:\localcache and return the path. If it is not found there, SymSrv will look in the Microsoft public symbol store at https://msdl.microsoft.com/download/symbols; if the file is found there, SymSrv will copy it to both \\interim\store and c:\localcache.

A similar behavior would be obtained by using the following path:

```
srv**\\interim\store*https://internetsite
```

In this case, the local cache is the default downstream store and the master store is an internet site. A mid-level store of \\interim\store has been specified for use in between the other two.

When SymSrv processes a path that contains cascading stores, it will skip any store that it cannot read or write to. So if a share goes down, it will copy the file to the store downstream from the missing store without any error. A nice side effect of this error is that the user can specify more than one master store that feeds a single stream of downstream stores as long as the master stores are not writable.

When a compressed symbol file is retrieved from the master store, it will be stored in compressed form in any mid-level store. The file will be uncompressed in the bottom-most store in the path.

Working With HTTP and SMB Symbol Server Paths

As previously discussed, chaining (or cascading) refers to the copy that occurs between each "*" separator in the symbol path. The symbols are searched for in a left-to-right order. On each miss, the next (upstream) symbol server is queried, until the file is found.

If found, the file is copied from the (upstream) symbol server to the previous (downstream) symbol server. This is repeated for each (downstream) symbol server. In this way, the (shared) downstream symbol servers are populated with the collective efforts of all clients using the symbol servers.

Even though chained UNC paths can be used without the SRV* prefix, we recommend that SRV* be specified so that the advanced error handling of symsrv.dll be used.

When including a HTTP symbol server in the path, only one can be specified (per chain), and it must be at the end of the path (as it can't be written to serve as a cache). If an HTTP-based symbol store was located in the middle or the left of the store list, it would not be possible to copy any found files to it and the chain would be broken. Furthermore, because the symbol handler cannot open a file from a web site, an HTTP-based store should not be the leftmost or only store on the list. If SymSrv is ever presented with this symbol path, it will attempt to recover by copying the file to the default downstream store and open it from there, regardless of whether the default downstream store is indicated in the symbol path or not.

HTTP is only supported when using the SRV* prefix (implemented by the symsrv.dll symbol handler).

Example HTTP and SMB Share Symbol Server Scenarios

A common UNC-only deployment involves a central office hosting all of the files (\\MainOffice\Symbols), branch offices caching a subset (\\BranchOfficeA\Symbols), and desktops (C:\Symbols) caching the files that they reference.

```
srv*C:\Symbols*\\BranchOfficeA\Symbols*\\MainOffice\Symbols
```

When the SMB share is the primary (upstream) symbol store, Read is required.

```
srv*C:\Symbols*\\MachineName\Symbols
```

When the SMB share is an intermediate (downstream) symbol store, Read/Change is required. The client will copy the file from the primary symbol store to the SMB share, and then from the SMB share to the local folder.

```
srv*C:\Symbols*\\MachineName\Symbols*https://msdl.microsoft.com/download/symbols
srv*C:\Symbols*\\MachineName\Symbols*\\MainOffice\Symbols
```

When the SMB share is an intermediate (downstream) symbol store in a SymProxy deployment, only Read is required. The SymProxy ISAPI Filter will perform the writes, not the client.

```
srv*C:\Symbols*\\MachineName\Symbols*https://SymProxyName/Symbols
```

Multiple HTTP and SMB Share Symbol Server Cache Scenarios

It is possible to specify multiple chains of symbol servers and cache locations, separated by a semi colon ";". If the symbols are located in the first chain, the second chain is not traversed. If the symbols are not located in the first chain, the second chain will be traversed and if the symbols are located in the second chain, they will be cached in the specified location. This approach will allow a primary symbol server to normally be used, with a secondary server only being used, if the symbols are not available on the primary symbol server specified in the first chain.

```
srv*C:\Symbols*\\Machine1\Symbols*https://SymProxyName/Symbols;srv*C:\WebSymbols*
https://msdl.microsoft.com/download/symbols
```

cachelocalsymbolcache***

Another way to create a local cache of symbols is by using the **cache*****localsymbolcache* string in your symbol path. This is not part of the symbol server element, but a separate element in your symbol path. The debugger will use the specified directory *localsymbolcache* to store any symbols loaded from any element that appears in your symbol path to the right of this string. This allows you to use a local cache for symbols downloaded from any location, not just those downloaded by a symbol server.

For example, the following symbol path will not cache symbols taken from *someshare*. It will use c:\mysymbols

to cache symbols taken from *anothershare*, because the element beginning with *anothershare* appears to the right of the **cache*c:\mysymbols** element. It will also use c:\mysymbols to cache symbols taken from the Microsoft public symbol store, because of the usual syntax used by the symbol server (**srv** with two or more asterisks). Moreover, if you subsequently use the **.sympath+** command to add additional locations to this path, these new elements will also be cached, since they will be appended to the right side of the path.

```
_NT_SYMBOL_PATH=\\someshare\that\cachestar\ignores;srv*c:\mysymbols*https://msdl.microsoft.com/download/symbol
s;cache*c:\mysymbols;\\anothershare\that\gets\cached
```

How SymSrv Locates Files

SymSrv creates a fully qualified UNC path to the desired symbol file. This path begins with the path to the symbol store recorded in the _NT_SYMBOL_PATH environment variable. The **SymbolServer** routine is then used to identify the name of the desired file; this name is appended to the path as a directory name. Another directory name, consisting of the concatenation of the *id*, *two*, and *three* parameters passed to **SymbolServer**, is then appended. If any of these values is zero, they are omitted.

The resulting directory is searched for the symbol file, or a symbol store pointer file.

If this search is successful, **SymbolServer** passes the path to the caller and returns **TRUE**. If the file is not found, **SymbolServer** returns **FALSE**.

Using AgeStore to Reduce the Cache Size

The AgeStore tool can be used to delete cached files that are older than a specified date, or to reduce the contents of the cache below a specified size. This can be useful if your downstream store is too large. For details, see AgeStore.

Firewalls and Proxy Servers

12/5/2018 • 2 minutes to read • Edit Online

If you are using SymSrv to access symbols, and your computer is on a network that uses a proxy server or the symbol store is outside your firewall, authentication may be required for data transmission to take place.

When SymSrv receives authentication requests, the debugger can either display the authentication request or automatically refuse the request, depending on how it has been configured.

SymSrv has integrated support for a proxy server. It can either use the default proxy server, SymProxy, or it can use another proxy server of your choice.

Authentication Requests

The debugger can be configured to allow authentication requests. When a firewall or proxy server requests authorization, a dialog box will appear. You will have to enter some sort of information (usually a user name and password) before the debugger can download symbols. If you enter incorrect information, the dialog box will be redisplayed. If you select the **Cancel** button, the dialog box will vanish and no symbol information will be transferred.

If the debugger is configured to refuse all authentication requests, no dialog box will appear, and no symbols will be transferred if authentication is required.

If you refuse an authentication request, or if the debugger automatically refuses an authentication request, SymSrv will make no further attempts to contact the symbol store. If you wish to renew contact, you must either restart the debugging session or use **!symsrv close**.

Note If you are using KD or CDB, the authentication dialog box may appear behind an open window. If this occurs, you may have to move or minimize some windows in order to find this dialog box.

In WinDbg, authentication requests are allowed by default. In KD and CDB, authentication requests are automatically refused by default.

To allow authentication requests, use either **!sym prompts** or **.symopt-0x80000**. To refuse all requests, use either **!sym prompts off** or **.symopt+0x80000**. To display the current setting, use **!sym**.

You must use **.reload (Reload Module)** after making any changes to the authentication permission status.

Choosing a Proxy Server

To select a default proxy server for Windows, open **Internet Options** in Control Panel, select the **Connections** tab, and then select the **LAN Settings** button. You can then enter the proxy server name and port number, or select **Advanced** to configure multiple proxy servers. For more details, see Internet Explorer's help file.

To select a specific proxy server for symsrv to use, set the _NT_SYMBOL_PROXY environment variable equal to the name or IP of the proxy server, followed by a colon and then the port number. For example:

```
set _NT_SYMBOL_PROXY=myproxyserver:80
```

When a proxy server is chosen in this way, it will be used by any Windows debugger that is using SymSrv to access a symbol server. It will also be used by any other debugging tool that uses DbgHelp as its symbol handler. No other programs will be affected by this setting.

HTTP Symbol Stores

9/25/2019 • 5 minutes to read • Edit Online

By using the SRV protocol supported through symsrv.dll (shipped with debugger), the symbol store can be accessed using HTTP (instead of just UNC/SMB).

HTTP is commonly used instead of SMB when a firewall doesn't allow SMB between the client and the server. Production and Lab environments are good examples of this.

An HTTP symbol server can't be a downstream store in a symbol path chain due to its read-only nature. Symbol Server Proxy (ISAPI Filter) works around this limit. SymProxy downloads the missing files to the server's file system using preconfigured upstream symbol stores. The filter downloads the file to the file system, allowing IIS to download the file to the client, thereby restoring the concept of symbol store chaining. Refer to SymProxy for more information.

Configuring IIS as a symbol store is relatively easy as the symbol files are just served as static files. The only non-default setting is the configuration of the MIME Types to allow the download of the symbol files as binary streams. This can be done by using a "*" wildcard applied to the virtual directory of the symbol folder.

In order to make a symbol store accessible over the Internet, you must configure both the directories containing the symbol files and Internet Information Services (IIS).

Note Because of the way IIS will be configured to serve symbol files, it is not recommended that the same server instance be used for any other purpose. Typically the desired security settings for a symbol server will not make sense for other uses, for example for an external facing commerce server. Make sure that the sample configuration described here makes sense for your environment and adapt it as appropriate for your specific needs.

Creating the Symbol Directory

Begin by selecting the directory you will use as your symbol store. In our examples, we call this directory c:\symstore and the name of the server on the network is \SymMachineName.

For details on how to populate your symbol store, see SymStore and Symbol Store Folder Tree.

Configuring IIS

Internet Information Services (IIS) must be configured to serve the symbols by creating a virtual directory and configuring MIME types. After this has been done, the authentication method may be chosen.

To create a virtual directory

1. From **Administrative Tools** open **Internet Information Services (IIS) Manager**.
2. Navigate to **Web Sites**.
3. Right-click **Default Web Site** or the name of the site being used and select **Add Virtual Directory...**.
4. Type **Symbols** for **Alias** and click **Next**.

   For ease of administration, it's recommended that the same name be used for the Folder, Share and Virtual Directory.
5. For the **Path** enter **c:\SymStore** and click **Next**.
6. Click **OK** to finish the adding the virtual directory.

Perform the subdirectory configuration process once for the server. Note that this is a global setting and will effect

applications not hosted in the root folder of a site.

Subdirectory Configuration

1. Navigate to **[Computer]**.
2. Open the **Configuration Editor**.
3. Navigate to **system ApplicationHost/sites**.
4. Expand **virtualDirectoryDefaults**.
5. Set **allowSubDirConfig** to *False*.

Perform this process once for the server. Note that this is a global setting and will effect applications not hosted in the root folder of a site.

Optionaly Make the Symbol Files Browseable

1. Navigate to **[Computer] | Sites | [Web Site] | Symbols**.
2. Double click **Directory Browsing** in the center pane.
3. Click **Enable** in the right pane.

The MIME Type for the downloaded content needs to be set to application/octet-stream to allow all symbols files to be delivered by IIS.

Configuring MIME types

1. Right-click the **Symbols** virtual directory and choose **Properties**.
2. Select **HTTP Headers**.
3. Click **MIME Types**.
4. Click **New**.
5. For **Extension**, type *****.
6. For **MIME type**, type **application/octet-stream**.
7. To exit the **MIME Types** dialog box, click **OK**.
8. To exit **Symbols Properties**, click **OK**.

You can edit the web.config file to configure MIME types for Symbols. This approach clears the inherited MIME Types and adds a catch-all wild card * MIME Type. This approach may be necessary when MIME types are being inherited in certain IIS configurations.

Using web.config to configure MIME types

1. Edit the web.config file as shown here.

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <system.webServer>
        <directoryBrowse enabled="true" />
        <staticContent>
            <clear />
            <mimeMap fileExtension=".*"
mimeType="application/octet-stream" />
        </staticContent>
    </system.webServer>
</configuration>
```

2. Restart IIS.

IIS is now ready to serve symbol files of all types from the symbol store.

Configuring Authentication

It is possible to configure IIS to use "Integrated Windows Authentication" so that clients (windbg.exe for example) can automatically authenticate against IIS without prompting the end-user for credentials.

Note Only configure Windows Authentication on IIS to control access to the symbol server if that is appropriate for your environment. There are other security options available to further control access to IIS if that is required for your environment.

To configure the authentication method as Anonymous

1. Launch the **Internet Information Services (IIS) Manager**.
2. Navigate to **[Computer] | Sites | [Web Site] | Symbols**.
3. Double click **Authentication** in the center pane.
4. Under **Authentication and access control** click **Edit**.
5. Right click **Windows Authentication** and select **Enable**.
6. For all other authentication providers, right click each provider and select **Disable**.
7. Click **OK** to finish configuring authentication.

If Window Authentication is not listed, use **Turn Windows features on and off** to enable the feature. The location of the feature is different in each version of Windows. In Windows 8.1/Windows 2012 R2, it is located under Internet Information Services | World Wide Web Services | Security.

Disable Kerberos Support

SymSrv.dll does not support Kerberos authentication when connecting to IIS. As such, Kerberos authentication must be disabled in IIS and NTLM needs to be set as the only Windows Authentication protocol.

Note Only disable Kerberos security if that is appropriate for your environment.

Disable Kerberos Support Using appcmd.exe

1. Open a Command Prompt window
2. To disable Kerberos and force the use of NTLM, use this command:

```
appcmd.exe set config -section:system.webServer/security/authentication/windowsAuthentication
/+"providers.[value='NTLM']" /commit:apphost
```

3. To return to the default value with Kerberos enabled, use this command:

```
appcmd.exe set config -section:system.webServer/security/authentication/windowsAuthentication
/+"providers.[value='Negotiate,NTLM']" /commit:apphost
```

Configuring SymSrv Client Authentication Prompts

When SymSrv receives authentication requests, the debugger can either display the authentication dialog box or automatically refuse the request, depending on how it has been configured. You can configure this behavior using !sym prompts on|off. For example to turn prompts on, use this command.

```
!sym prompts on
```

To check the current setting, use this command.

```
!sym prompts
```

For more information see **!sym** and Firewalls and Proxy Servers.

File Share (SMB) Symbol Server

12/5/2018 • 2 minutes to read • Edit Online

Running a SMB Symbol Server is simply a matter of creating a file share and granting users access to that file share.

Creating a SMB File Share Symbol Store

Use Windows Explorer or Computer Management to create the File Share and assign security. These steps assume that the symbols will be located in *D:\SymStore\Symbols*. Complete these steps using Windows Explorer:

1. Open **Windows Explorer**.
2. Right-click *D:\SymStore\Symbols* and choose **Properties**.
3. Click on the **Sharing** tab.
4. Click on **Advanced Sharing**... .
5. Check *Share this folder*.
6. Click on **Permissions**.
7. Remove the *Everyone* group.
8. Using **Add...**, add the Users/Security Groups requiring access.
9. For each User/Security Group added, grant Read or Read/Change access.
10. Click on **OK** (Permissions dialog).
11. Click on **OK** (Advanced Sharing dialog).
12. Press **Close** (Properties dialog).

Complete these steps using Computer Management:

1. Type *Computer* in Window Start (resolves as This PC in Windows 8).
2. Right-click and select *Manage*.
3. Navigate to *System Tools* | *Shared Folders* | *Shares*.
4. Right-click and select **New | Share...** .
5. Press **Next** (Create a Shared Folder Wizard dialog).
6. Enter **D:\SymStore\Symbols** as the Folder Path.
7. Press **Next** twice.
8. Select **Customize permissions**.
9. Press **Custom...** .
10. Remove *Everyone*.
11. Using **Add...**, add the Users/Security Groups requiring access.
12. For each User/Security Group added, grant Read or Read/Change access.

13. Press **OK** (Customize Permissions dialog).
14. Press **Finish** twice to complete the process.

Test The SMB File Share

Configure a debugger to use this symbol path:

```
srv*C:\Symbols*\\MachineName\Symbols
```

To view the location of the PDBs being referenced in the debugger, use the lm (list modules) command. The path to the PDBs should all begin with C:\Symbols. By running "!sym noisy", and ".reload /f", you will see extensive symbol logging of the download of the symbols and images from the \\MachineName\Symbols file server to C:\Symbols.

File Share Symbol Path

There are multiples ways to configure your debugger's symbol path (.sympath) to use a File Share. The syntax of the symbol path determines if the symbol file will be cached locally or not, and where it is cached.

Direct File Share use (no local caching):

```
srv*\\MachineName\Symbols
```

Local Caching of the File Share's files to a particular local folder (e.g. c:\Symbols):

```
srv*c:\Symbols*\\MachineName\Symbols
```

Local Caching of the File Share's files to the %DBGHELP_HOMEDIR%\Sym folder:

```
srv**\\MachineName\Symbols
```

The second "*" in the example shown above, represents the default local server cache.

If the DBGHELP_HOMEDIR variable is not set, DBGHELP_HOMEDIR defaults to the debugger executable folder (for example C:\Program Files\Windows Kits\10.0\Debuggers\x86) and causes caching to occur in C:\Program Files\Windows Kits\10.0\Debuggers\x86\Sym.

Related topics

Symbol Store Folder Tree

Symbol Store Folder Tree

12/5/2018 • 2 minutes to read • Edit Online

The symbol store backing SMB and HTTP requests is a folder tree residing on a local disk.

To keep administration simple, the sub-folder name (e.g. Symbols) can also be used as the File Share name and also the Virtual Directory name. If a new symbol store was to be added, a new sub-folder would be made under D:\SymStore, and a new File Share and Virtual Directory of that name would be made to expose the store to clients.

The folder tree's location should be chosen carefully as well as the disk's file system. The symbol store can get extremely big (terabytes) when caching files from (internal) build servers and the Internet. The folder tree should reside on a disk that is capable of a high number of reads and low number of writes. The file system can affect performance - ReFS may perform better than NTFS and should be investigated for large deployments. Equally, the networking to the server should be of sufficient speed to handle the load from the clients and also the load to the upstream symbol stores to retrieve the symbols for cache population.

Symbol Store Single-Tier or Two-Tier Structure

Normally files are placed in a single tier directory structure in which a single subdirectory exists for each filename cached. Under each filename folder, additional folders are made to store each version of the file. The tree will have this structure:

```
D:\SymStore\Symbols\ntdll.dll\...\
D:\SymStore\Symbols\ntdll.pdb\...\
D:\SymStore\Symbols\kernel32.dll\...\
D:\SymStore\Symbols\kernel32.pdb\...\
```

If a large number of files are to be stored, a two-tier structure can be used at the root of the symbol store. The first 2 letters of the filename are used as an intermediate folder name.

To use a two-tier structure, place a file called index2.txt in the root of D:\SymStore\Symbols. The content of the file is of no importance. When this file exists, symsrv.dll will create and consume files from the two-tier tree using this structure:

```
D:\SymStore\Symbols\nt\ntdll.dll\...\
D:\SymStore\Symbols\nt\ntdll.pdb\...\
D:\SymStore\Symbols\ke\kernel32.dll\...\
D:\SymStore\Symbols\ke\kernel32.pdb\...\
```

If you want to convert the structure after the symbol store is populated, use the convertstore.exe application in the debugger folder. To allow the tool to work, create a folder called 000Admin in the root folder. This folder is required by convertstore.exe so that it can control the locking of the symbol store.

Related topics

HTTP Symbol Stores

File Share (SMB) Symbol Server

SymProxy

1/9/2019 • 2 minutes to read • Edit Online

You can configure your HTTP-based symbol store to act as a proxy between client computers and other symbol stores. The implementation is through an Internet Server Application Programming Interface (ISAPI) filter called SymProxy (Symproxy.dll). The SymProxy server can be used as a gateway computer to the Internet or other sources within your company network. The following diagram shows an example SymProxy configuration.



SymProxy is useful in many situations. For example:

- You are debugging many systems within a lab environment in which the computers are not attached to the company network, but the symbols are stored in the network and must be accessed using Integrated Windows Authentication (IWA).
- Your corporate computing environment includes a firewall that prevents access to the Internet from computers that are debugging and you must obtain symbols from an internet Web site.
- You want to present a single symbol path for all users in your company so that they need not know or care about where symbols are located, and you can add new symbol stores without user intervention.
- You have a remote site that is physically far from the rest of your company resources, and network access is slow. This system can be used to acquire symbols and cache them to the remote site.

To install SymProxy, you must manually copy the files to the correct location, configure the registry, choose network security credentials, and configure Internet Information Services (IIS). To ensure that your HTTP symbol store is properly configured, see HTTP Symbol Stores.

Multiple Symbol Server Performance Considerations

Each Virtual Directory can be associated with multiple (upstream) symbol stores. Each symbol store is queried independently. For performance, local SMB servers should be processed before internet HTTP servers. Unlike a debugger symbol path, multiple HTTP symbol stores can be specified in a SymProxy symbol path. A maximum of 10 entries are supported per Virtual Directory.

SymProxy Symbol Path

SymProxy splits the (registry defined) symbol path value up in to the individual entries and uses each entry to generate a SRV* based symbol path to retrieve the file. It uses the Virtual Directory's folder as the downstream store in each of the queries – in effect, merging the upstream stores in to a single downstream symbol store.

The (generated) symbol path used by SymProxy is equivalent to this:

```
SRV*<Virtual Directory Folder>*<SymbolPath Entry #N>
```

In this example, a UNC path and two HTTP paths are associated with a Virtual Directory to merge the symbols from a corporate symbol server, Microsoft and a 3rd party (Contoso). The SymProxy SymbolPath would be set like this:

```
\\MainOffice\Symbols;https://msdl.microsoft.com/download/symbols;
https://symbols.contoso.com/symbols
```

The Main Office Symbol file share is queried first using a (generated) symbol path of:

```
SRV*D:\SymStore\Symbols*\\MainOffice\Symbols
```

If the symbol file is not found, the Microsoft Symbol Store is queried using a (generated) symbol path of:

```
SRV*D:\SymStore\Symbols*https://msdl.microsoft.com/download/symbols
```

If the file is still not found, the Contoso Symbol Store (https://symbols.contoso.com/symbols) is queried using a (generated) symbol path of:

```
SRV*D:\SymStore\Symbols*https://symbols.contoso.com/symbols
```

This section includes:

Installing SymProxy

Configuring the Registry

Choosing Network Security Credentials

Configuring IIS for SymProxy

Setting Up Exclusion Lists

Dealing with Unavailable Symbol Stores

Handling File Pointers

Caching Acquired Symbol Files

Installing SymProxy

3/14/2019 • 2 minutes to read • Edit Online

Summary of installation tasks

The following summarizes the tasks to install and configure SymProxy.

- The SymProxy files need to be copied to the %WINDIR%\system32\inetsrv folder for IIS. This task is discussed below.
- The registry needs to be configured for SymProxy. For more information see Configuring the Registry.
- The manifest needs to be registered as Performance Counters and ETW events, and the Event Log needs to be configured.
- IIS needs to be configured. For more information, see Choosing Network Security Credentials and Configuring IIS for SymProxy.

These steps can be automated using the Install.cmd file. For more information, see SymProxy Automated Installation.

Copy the SymProxy files to IIS

The SymProxy files are included in the Debuggers directory of the Windows Driver Kit. For example this is the location of the 64 bit files for Windows 10 kit. C:\Program Files (x86)\Windows Kits\10\Debuggers\x64\symproxy.

To install SymProxy on the server, copy symproxy.dll, symsrv.dll and symproxy.man to %WINDIR%\system32\inetsrv.

In order to prevent problems that could occur in accessing the Microsoft Symbol Store, create a blank file called %WINDIR%\system32\inetsrv\symsrv.yes. The contents of this file are not important. When symsrv.yes file is present, it automatically accepts the EULA for the Microsoft Public Symbol Store.

Note that the certificates that are normally installed with IIS and Windows server such as the "Baltimore CyberTrust Root" are used for HTTPS/TLS communication to the upstream provider, and they need to be in the Trusted Root store on the machine where SymProxy is running. For general information on troubleshooting SSL issues, see Troubleshooting SSL related issues (Server Certificate).

Configuring the Registry

7/1/2019 • 8 minutes to read • Edit Online

SymProxy stores its settings in this registry key.

```
HKLM/Software/Microsoft/Symbol Server Proxy
```

This registry key controls the location from which to find symbols to store in the Web site, the logging level, and whether or not SymProxy operates with a direct connection to the network. You can create this key by running the SymProxy registration tool (Symproxy.reg) provided with Debugging Tools for Windows. Type **symproxy.reg** at the command prompt or double-click it from Windows Explorer.

This will add entries for the settings that will be prefixed with an "x" so that they are disabled. To enable a setting, remove the "x" from in front of the desired setting.

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Symbol Server Proxy]
"Available Settings"="Remove the 'x' prefix to use the setting"
"xMissAgeTimeout"=dword:00015180
"xMissAgeCheck"=dword:00000e10
"xMissTimeout"=dword:00000e10
"xNoCache"=dword:00000001
"xNoFilePointers"=dword:00000001
"xNoInternetProxy"=dword:00000001
"xNoLongerIndexedAuthoritive"=dword:00000001
"xNoUncompress"=dword:00000001
"xRequestTimeout"=dword:0000019
"xRetryAppHang"=dword:0000002
"xUriFilter"=dword:00000FF
"xUriTiers"=dword:0000001
```

The symproxy.reg registry file assumes a virtual directory name of Symbols and configures the Symbol Path to use the Microsoft Public Symbol Server.

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Symbol Server Proxy\Web Directories]

[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Symbol Server Proxy\Web Directories\Symbols]
"SymbolPath"="https://msdl.microsoft.com/download/symbols"
```

The event logging entries in symproxy.reg are covered latter in the Event Log section of this topic.

```
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\EventLog\Application\Microsoft-Windows-SymProxy]
"ProviderGuid"="{0876099c-a903-47ff-af14-52035bb479ef}"
"EventMessageFile"=hex(2):25,00,53,00,79,00,73,00,74,00,65,00,6d,00,52,00,6f,\
  00,6f,00,74,00,25,00,5c,00,73,00,79,00,73,00,74,00,65,00,6d,00,33,00,32,00,\
  5c,00,69,00,6e,00,65,00,74,00,73,00,72,00,76,00,5c,00,53,00,79,00,6d,00,50,\
  00,72,00,6f,00,78,00,79,00,2e,00,64,00,6c,00,6c,00,00,00
"TypesSupported"=dword:00000007
```

The web directory entries in symproxy.reg are discussed in this topic.

Web Directories

For each virtual directory generated in IIS that you are using as a symbol store, you must setup a registry key below the **Web Directories** subkey of the following registry key.

```
HKLM/Software/Microsoft/Symbol Server Proxy
```

To edit the registry key for a symbol store virtual directory

- Edit the contents of **SymbolPath** to contain all of the symbol stores used by the SymProxy symbol store. If there is more than one symbol store being used, separate them with semicolons. A maximum of 10 stores is supported for each value. HTTP paths must include the **https:// prefix**, and UNC paths must include the **** prefix.

For example, if one of the virtual directories is called Symbols, and the symbols stores that it accesses are located at the UNC store \\symbols\symbols and the HTTP store https://msdl.microsoft.com/download/symbols, create the following registry key.

```
HKLM/Software/Microsoft/Symbol Server Proxy/Web Directories/Symbols
```

After this key is created, edit its **SymbolPath** to be \\symbols\symbols;https://msdl.microsoft.com/download/symbols. This can be seen in the following screenshot of the Registry Editor.



In this example, SymProxy first searches for symbols in \\symbols\symbols. If the files are not found there, the Microsoft Symbol Store will be used.

- In each of the keys under Web Directories that match the Virtual Directory names, a REG_SZ called SymbolPath needs to be created. The value contains all the upstream symbol stores that will be used to populate the SymProxy symbol store.
- A maximum of 10 entries are supported.
- Separate entries with semicolons.
- UNC paths need to include the "\\" prefix
- HTTP paths need to include the "https://" prefix
- Order the values from least expensive to most expensive.
    - You will need to balance usage performance goals vs. server and data communications costs in the calculation.
    - In general, put local SMB/HTTP servers before internet HTTP servers.

SymProxy Performance Counters

SymProxy can emit performance counters via a provider called SymProxy.

To enable the performance counters support, register the symproxy manifest file in an administrator command window:

```
C:\> lodctr.exe /m:%WINDIR%\system32\inetsrv\symproxy.man
```

To disable the performance counters support, unregister the manifest:

```
C:\> unlodctr.exe /m:%WINDIR%\system32\inetsrv\symproxy.man
```

SymProxy Event Tracing for Windows

SymProxy can create ETW events via a provider called Microsoft-Windows-SymProxy.

```
C:\> logman query providers | findstr SymProxy
Microsoft-Windows-SymProxy {0876099C-A903-47FF-AF14-52035BB479EF}
```

To enable the ETW support, register the manifest file:

```
C:\> wevtutil.exe install-manifest %WINDIR%\system32\inetsrv\symproxy.man
```

To disable the ETW support, unregister the manifest file:

```
C:\> wevtutil.exe uninstall-manifest %WINDIR%\system32\inetsrv\symproxy.man
```

Event Log

If ETW is configured, the events are recorded as events in the *Operational and Analytic* channels under *Applications and Services Logs\Microsoft\Windows\SymProxy* in the Event Log.

To correctly view the message of the Event Log entries, the Event Log area of the symproxy.reg file needs to be added to the registry:

```
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\EventLog\Application\Microsoft-Windows-SymProxy]
"ProviderGuid"="{0876099c-a903-47ff-af14-52035bb479ef}"
"EventMessageFile"=hex(2):25,00,53,00,79,00,73,00,74,00,65,00,6d,00,52,00,6f,\
  00,6f,00,74,00,25,00,5c,00,73,00,79,00,73,00,74,00,65,00,6d,00,33,00,32,00,\
  5c,00,69,00,6e,00,65,00,74,00,73,00,72,00,76,00,5c,00,53,00,79,00,6d,00,50,\
  00,72,00,6f,00,78,00,79,00,2e,00,64,00,6c,00,6c,00,00,00
"TypesSupported"=dword:00000007
```

SymProxy Events

SymProxy logs the following events:

Event ID	**Description**	**Channel**
1	Start of the ISAPI filter	Admin
2	Stop of the ISAPI filter	Admin
3	Configuration of the ISAPI filter	Admin
4	Miss Cache Statistics	Admin
10	URL Request - Local Cache Hit	Operational

11	URL Request - Local Cache Miss	Operational
20	Symbol Download via SymSrv	Operational
30	Critical Symbol Missing	Admin
31	Critical Image Missing	Admin
40	SymSrv – Path Not Found	Admin
41	SymSrv – File Not Found	Admin
42	SymSrv – Access Denied	Admin
43	SymSrv – Path Too Long	Admin
49	SymSrv – Error Code	Admin
90	Lock Contention	Operational
100	General Critical Message	Analytic
101	General Error Message	Analytic
102	General Warning Message	Analytic
103	General Informational Message	Analytic
104	General Analytic Message	Analytic
105	General Debug Message	Debug

Symbol Server Proxy Configuration

SymProxy stores its configuration settings in the following registry key area:

```
HKLM/Software/Microsoft/Symbol Server Proxy
```

From this location, SymProxy acquires its global settings and the symbol paths of upstream symbol stores.

You can create this key by merging in the symproxy.reg file you customized as discussed earlier.

Symbol Server Proxy Key

The Symbol Server Proxy registry key supports the following global settings (all REG_DWORD). Settings can be applied live by recycling the application pool. A new w3wp.exe process will be created and it will read the new values. Once all pending requests to the old w3wp.exe process have completed, the old w3wp.exe process will end. IIS by default recycles w3wp.exe processes every 1,740 minutes (29 hours).

REG_DWORD	Description

NoInternetProxy	When running as a service, SymSrv.dll uses WinHTTP instead of WinInet to make HTTP requests. Consequently, you may need to set up HTTP proxy settings so that the service can access outside network resources. You can do this using the netsh program. Type “netsh.exe winhttp -?” for instructions. By default, SymProxy uses the designated HTTP proxy. If no HTTP proxy is configured, SymProxy will use a dummy proxy. This allows secure access to HTTP sites within your intranet. As a side effect, this prevents SymProxy from directly connecting to non-secure sites. Creating the REG_DWORD:"NoInternetProxy" value configures SymProxy to operate without a proxy, allowing a direct connection.
NoFilePointers	By default, for symbols that don’t exist, SymProxy will look for a file.ptr file next to the requested file (in the local cache). If found, it will return the location specified by the file.ptr file. This ability is only required when the local cache is being populated by SymStore.exe. Create the REG_DWORD:"NoFilePointers" value to skip the lookup.
NoUncompress	By default, SymProxy will decompress downloaded symbols before returning the file to the caller. This reduces CPU at the client, but increases I/O. Create the REG_DWORD:"NoUncompress" value to skip the decompression.
NoCache	By default, SymProxy will cache downloaded symbols to the local file system, defined by the virtual directory’s path. Create the REG_DWORD:"NoCache" value to skip the download and to provide the remote path of the file to the client instead.

MissTimeout	Timeout period, in seconds, for which missing symbols are reported as missing without re-querying the upstream symbol servers. A miss is associated with a UTC based time. Subsequent requests for the file are immediately rejected for N seconds. The first request for the file after N seconds causes the upstream symbol stores to be re-queried. On success, the symbol file is returned and the miss is deleted. On failure, the miss is moved forward to the current time (in UTC) to start a new timeout period. Use the "Miss Cache " *counters to monitor the misses*. • *Unspecified - (default) 300 seconds/5 minutes* • *0 – Feature disabled* • *N – Timeout lasts N seconds*
MissAgeCheck	Period between Miss Age checks. The Miss cache is scanned and records older than MissAgeTimeout seconds are removed. The current statistics are saved to the Event Log using Event ID 4. • Unspecified - (default) 3600 seconds / 1 hour • 0 – Feature disabled • N – Period between checks in N seconds
FailureTimeout FailureCount FailurePeriod FailureBlackout	The Blackout feature is used to termporarly disable upstream symbol stores that are unresponsive. The Blackout feature uses 4 REG_DWORD values to define the behaviour. By default, the feature is disabled. For each upstream symbol store defined in a Symbol Path, failures are individually recorded. If a request takes longer than FailureTimeout (msec), the failure count is incremented. The Symbol Path is marked as dead after FailureCount failures in FailurePeriod seconds. At this time, all requests are ignored until FailureBlackout seconds have elapsed. The first caller after the timeout tests the upstream symbol store. On success, the timeout is removed and requests are allowed. On failure, the time is set to Now+FailureBlackout seconds. After that time, the upstream symbol store is tested again.
MissAgeTimeout	Description Pending
RequestTimeout	Description Pending
RetryAppHang	Description Pending

NoLongerIndexedAuthoritive	Description Pending
UriFilter	Description Pending
UriTiers	Description Pending

Accessing Outside Network Resources

When SymSrv is used in conjunction with SymProxy, it runs as a service and uses the WinHTTP API to access symbols over an HTTP connection. This differs from its usual behavior of using WinInet for this purpose.

Consequently, you may need to set up HTTP proxy settings so that this service can access outside network resources. Use one of the following methods to configure these settings:

- Use the Netsh tool (netsh.exe). For instructions, type the following in a Command Prompt window:

```
netsh winhttp -?
```

The default behavior of SymProxy is to use whatever HTTP proxy is designated by either ProxyCfg or Netsh. If no HTTP proxy is configured, SymProxy uses a dummy proxy to allow access to secure HTTP sites within your intranet. As a side effect, this technique prevents SymProxy from working with direct connections to the external Internet. If you wish to permit SymProxy to operate with a direct connection to the Internet, create a REG_DWORD value named **NoInternetProxy** in the **Symbol Server Proxy** key of your registry. Set the value of **NoInternetProxy** to 1 and verify that there is no HTTP proxy indicated by ProxyCfg.

Choosing Network Security Credentials

12/5/2018 • 2 minutes to read • Edit Online

The symbol proxy server must run from a security context with the appropriate privileges for access to the symbol stores that you plan to use. If you obtain symbols from an external Web store such as https://msdl.microsoft.com/download/symbols, the symbol proxy server must access the Web from outside of any firewalls. If you obtain files from other computers on your network, the symbol proxy server must have appropriate privileges to read files from those locations. Two possible choices are to set the symbol proxy server to authenticate as the **Network Service** account or to create a user account that is managed within Active Directory Domain Services along with other user accounts.

Note It is a good practice to limit privileges of this account to only those necessary to read files and copy them to c:\symstore. This restriction prevents clients that access your HTTP store from corrupting the system.

Note Make sure the options presented here make sense in your environment. Different organizations have different security needs and requirements. Modify the process outlined here to support the security requirements of your organization.

Authenticate as a Network Service

The **Network Service** account is built in to Windows, so there is no extra step of creating a new account. For this example, we name the computer where the symbol proxy server is being configured *SymMachineName* on a domain named *corp*.

External symbol stores or Internet proxies must be configured to allow this computer's **Network Service** account (Machine Account) to authenticate successfully. There are two ways to achieve this:

- Allow access to the **Authenticated Users** group on the external store or Internet proxy.
- Allow access to the Machine Account *corp\SymMachineName$*. This option is more secure because it limits access to just the symbol proxy server's "Network Service" account.

Authenticate as a Domain User

For this example, we will presume the user account is named *SymProxyUser* on a domain called *corp*.

To add the user account to the IIS_USRS group

1. From **Administrative Tools** open **Computer Management**.
2. Expand **Local Users and Groups**.
3. Click **Groups**.
4. Double-click **IIS_USRS** in the center pane and select **Properties**.
5. Under the **Members** section, click **Add**.
6. Type *corp\SymProxyUser* in the pane labeled **Enter the object name to select**.
7. To exit the **Select Users, Computer, or Groups** dialog box, click **OK**.
8. To exit **IIS_USRS Properties**, click **OK**.
9. Close the **Computer Management** console.

Set up IIS to use the account

1. From **Administrative Tools** open **Internet Information Services (IIS) Manager**.
2. Expand **Web Sites**.
3. Right click **Default Web Site** and choose **Properties**.
4. Click the **Directory Security** tab.
5. In the **Authentication and access control** section, click **Edit...**.
6. Make sure that *Enable anonymous access* is checked.
7. Enter the credentials of the account that has permissions to access the remote symbol server store(s) ("corp\SymProxyUser") , then click **OK**.
8. Re-enter the password when asked and click **OK**.
9. To exit **Default Web Site Properties**, click **OK**.
10. You may be presented with the *Inheritance Overrides* dialog. If so, select which virtual directories you want to have this apply to.

Authenticate as a Domain User Using the IIS_WPG group

For this example, the user account is named *SymProxyUser* on a domain named *corp*. To authenticate this user account, it must be added to the **IIS_WPG** group.

To add the user account to the IIS_WPG group

1. From **Administrative Tools** open **Computer Management**.
2. Expand **Local Users and Groups**.
3. Click **Groups**.
4. Double-click **IIS_WPG** in the right pane.
5. Click **Add**.
6. Type *corp\SymProxyUser* in the pane labeled **Enter the object name to select**.
7. To exit the **Select Users, Computer, or Groups** dialog box, click **OK**.
8. To exit **IIS_WPG Properties**, click **OK**.
9. Close the **Computer Management** console.

Configuring IIS for SymProxy

12/5/2018 • 2 minutes to read • Edit Online

Internet Information Services (IIS) must be configured to use SymProxy as an Internet Server Application Programming Interface (ISAPI) filter. Furthermore, permissions must be set so that IIS can obtain symbols.

To configure the application pool

1. From **Administrative Tools** open **Internet Information Services (IIS) Manager**.
2. Expand the entry with the computer name on the left and locate **Application Pools**.
3. Right-click **Application Pools** and choose **Add Application Pool**.
4. For the **Name** type *SymProxy App Pool*.
5. Under **.Net Framework version** select *None*
6. Click **OK** to create the application pool.
7. Next, right click the entry for the new application pool and select **Advanced Settings...**.
8. Under **Process Model**, you will see **Identity**. Click the button at the right labeled "**...**".
    a. If you are authenticating as a network service, select **Predefined** for the **Application Pool Identity** and then select **Network Service**.
    b. If you are authenticating as a domain user, select **Custom account** and then click the **Set** button. Type the credentials of the account that has permissions to access the remote symbol server store (for example, *corp\SymProxyUser*), and click **OK**.
9. Click **OK** to exit the **Application Pool Identity** dialog.
10. Click **OK** to exit the **Advanced Settings** dialog.

To set up the Virtual Directory

1. Expand **Web Sites**.
2. Select **Default Web Site**.
3. Right-click the **Symbols** virtual directory and choose **Properties**.
4. In the **Virtual Directory** tab, click **Create**.
5. From the **Application Pool** drop-down menu, choose **SymProxy App Pool**.
6. To exit **Symbols Properties**, click **OK**.

Configure the ISAPI Filter

1. Right-click the **Default Web Site** and select **Properties**.
2. Double-click **ISAPI Filters**.
3. Right click the center pane under the column **Name** and select Click **Add**.
4. For **Filter Name** type **SymProxy** or some other meaningful name.
5. For **Executable** type *c:\windows\system32\inetsrv\symproxy.dll*.

6. To exit the **Filter Properties** dialog, click **OK**.
7. To exit **Default Web Site Properties,** click **OK**.

Setting Up Exclusion Lists

12/5/2018 • 2 minutes to read • Edit Online

In some environments, you may find yourself debugging systems that have a large quantity of modules loaded for which you cannot obtain symbols. This is often the case if you have code that is called by a third-party vendor. This can result in a lot of failed attempts to find symbols, which is time-consuming and clogs up network resources. To alleviate this situation, you can use an *exclusion list* to specify symbols that should be excluded from the search. This feature exists in the client debugger, but you can also configure the SymProxy filter to use its own the exclusion list and prevent such network activity where it is most likely to take up resources.

The exclusion list is made up of the names of the files for which you want to prevent processing. The file names can contain wildcards. For example:

```
dbghelp.pdb
symsrv.*
mso*
```

The list can be implemented in two ways. The first is in an .ini file, %WINDIR%\system32\inetsrv\Symsrv.ini. A section called "exclusions" should contain the list:

```
[exclusions]
dbghelp.pdb
symsrv.*
mso*
```

Alternatively, you can store the exclusions in the registry. Create a key named

```
HKLM\ Software\Microsoft\Symbol Server\Exclusions
```

Store the file name list as string values (REG_SZ) within this key. The name of the string value acts as the file name to exclude. The contents of the string value can be used as a comment describing why the file is being excluded.

SymProxy reads from the exclusion list every half-hour so that you do not need to restart the Web service to see changes take effect. Add files to the list in the registry or .ini file and wait a short period for the exclusions to be used.

Note SymProxy does not support the use of both Symsrv.ini and the registry. If the .ini file exists, it is used. Otherwise, the registry is checked.

Dealing with Unavailable Symbol Stores

12/5/2018 • 2 minutes to read • Edit Online

If one of the symbol stores that SymSrv is configured to obtain files from is down or otherwise unavailable, the result can be long waits from the client for every file request. When SymSrv is called from SymProxy, you can avoid most of these waits by setting up SymSrv to stop trying to access the store in question. When this feature is engaged, SymSrv stops trying to use the store for a set period of time after it experiences a specified number of timeouts from the same store during a set interval. The values of these variables can be controlled either by an .ini file or from the registry.

To control symbol store access using a .ini file

1. In %WINDIR%\system32\inetsrv\Symsrv.ini, create a section called **timeouts**.
2. Add the values **trigger**, **count**, and **blackout** to this section.

Trigger indicates the amount of time in minutes to watch for timeouts. **Count** indicates the number of timeouts to look for during the **trigger** period. **Blackout** indicates the length of time in minutes to disable the store after the threshold is reached.

For example, we recommend the following settings:

```
[timeouts]
trigger=10
count=5
blackout=15
```

In this example, the store access is turned off if five timeouts are experienced in a 10-minute period. At the completion of a 15-minute blackout, the store is reactivated.

To control symbol store access using the registry

1. Create a key named

   ```
   HKLM\ Software\Microsoft\Symbol Server\Timeouts
   ```

2. Add three REG_DWORD values **trigger**, **count**, and **blackout** to this key. Set these values as you would in the .ini file.

Whether using the registry or an .ini file, if any of the trigger, count, or blackout values are set to 0 or if any of the keys or values do not exist, this functionality is disabled.

This feature of SymSrv is currently available only when running as a service. This means that the only practical application of this feature is when it is called from SymProxy.

Handling File Pointers

12/5/2018 • 2 minutes to read • Edit Online

A UNC symbol store supports placing the actual files to be served in a separate location, with the client code finding the location of the files through file pointers. These pointers are generated in the symbol store using SymStore with the **/p** option. This handling is supported with other HTTP-based symbol stores only if the file pointers point to a UNC location that is directly accessible by the client. When SymProxy is loaded into the Web server, file-pointer handling is automatically enhanced. The client no longer needs to be able to directly access the target files because SymProxy serves them through the HTTP interface.

Because this feature is automatically applied, an option exists to turn it off in case you must use the proxy for serving some files and regular file pointer implementation for others. To do this, create a REG_DWORD called "NoFilePointerHandler" in **HKLM\Software\Microsoft\Symbol Server**. Set this value to 1 (or anything other than 0) to turn off the internal file pointer handler in SymProxy.

Caching Acquired Symbol Files

12/5/2018 • 2 minutes to read • Edit Online

Typically, SymProxy caches the files that it acquires in the directory designated within Internet Information Services (IIS) as the virtual root for the associated Web site. Then IIS makes the file available to the client debugger. Because the debugger cannot open a file directly from HTTP, it copies the file to a local cache, specified by the symbol path:

```
srv*c:\localcache*https://server/symbols
```

In this example, the client debugger copies the file to c:\localcache. In a situation such as this, the file is copied twice - once by SymProxy to the virtual root of the Web site, and again by the debugger to its local cache.

It is possible to avoid the second copy operation and speed up processing. To do this, you must first share the virtual root of the Web site as a UNC path that can be accessed by the debuggers. For sake of example, this path is named \\server\symbols. You must then remove the IIS configuration for MIME types:

To remove the IIS configuration for MIME types

1. From **Administrative Tools** open **Internet Information Services (IIS) Manager**.
2. Expand **Web Sites**.
3. Right-click **Default Web Site**.
4. Right-click the **Symbols** virtual directory and select **Properties**.
5. Click the **HTTP Headers** tab.
6. Click **MIME Types** .
7. Select all types in the list box labeled **Registered MIME Types**.
8. Click **Remove** .
9. To exit the **MIME Types** dialog, click **OK**.
10. To exit **Symbols Properties**, click **OK**.

This causes IIS to return **file not found** to the debugging client for all transactions on the Web site. However, it does not prevent SymProxy from populating the virtual root with the file.

After you remove the IIS configuration for MIME types, configure the debugger clients to look for symbols first in the HTTP store and in the share that maps to the virtual root of the store with the command:

```
srv**https://server/symbols;srv*\\server\symbols
```

In the preceding example, the first element of the symbol path (srv**https://server/symbols) says to get files from the HTTP store and copy them to the default symbol store as a local cache. The specified cache is of no importance because no file is ever received from the HTTP store. After this failure, it attempts to obtain the file from the actual location of the virtual root of the store (srv*\\server\symbols). This attempt succeeds because the file is copied to that location as a side effect of the previous path processing.

SymProxy Automated Installation

3/14/2019 • 2 minutes to read • Edit Online

These steps along with the Install.cmd script below can help automate the installation of SymProxy to a default IIS installation. You will likely need to adapt these steps to the specific needs of your environment.

1. Create D:\SymStore\Symbols folder.
   - Grant Read to Everyone
   - Grant Read\Write to the SymProxy App Pool user account (Domain\User)
2. Share D:\SymStore\Symbols as Symbols.
   - Grant Read to Everyone (or be more specific)
3. (Optionally) Create an empty file called index2.txt in D:\SymStore\Symbols.
4. (Optionally) Create an empty file called %WINDIR%\system32\inetsrv\symsrv.yes. This accepts the EULA for the Microsoft Public Symbol Store.
5. Determine the parameters for Install.cmd and run it.
6. Configure the clients symbol path using the server name that you created.

```
SRV*\\MachineName\Symbols*https://MachineName/Symbols
```

The Install.cmd script requires 3 parameters:

- Virtual Directory path (e.g. D:\SymStore\Symbols)
- Username (for the Application Pool)
- Password (for the Application Pool)

To clear the MIME Type inheritance, an XML file is needed to drive the associated AppCmd.exe command. Place the staticContentClear.xml file shown below in the same folder as the Install.cmd script to achieve this result.

Example Install.Cmd parameter usage:

```
Install.cmd D:\SymStore\Symbols CONTOSO\SymProxyService Pa$$word
```

Install.cmd

```
@echo off

SET VirDirectory=%1
SET UserName=%2
SET Password=%3

::
::  SymProxy dll installation.
::

copy symproxy.dll %windir%\system32\inetsrv
copy symproxy.man %windir%\system32\inetsrv
copy symsrv.dll %windir%\system32\inetsrv

lodctr.exe /m:%windir%\system32\inetsrv\symproxy.man
wevtutil.exe install-manifest %windir%\System32\inetsrv\symproxy.man
regedit.exe /s symproxy.reg

::
::  Web server Configuraiton
::

IF not exist %VirDirectory% mkdir %VirDirectory%

rem Make the 'Default Web Site'
%windir%\system32\inetsrv\appcmd.exe add site -site.name:"Default Web Site" -bindings:"http/*:80:" -
physicalPath:C:\inetpub\wwwroot

rem Enabled Directory Browsing on the 'Default Web Site'
%windir%\system32\inetsrv\appcmd.exe set config "Default Web Site" -section:system.webServer/directoryBrowse
/enabled:"True"

rem Make the 'SymProxy App Pool'
%windir%\system32\inetsrv\appcmd.exe add apppool -apppool.name:SymProxyAppPool -managedRuntimeVersion:
%windir%\system32\inetsrv\appcmd.exe set apppool -apppool.name:SymProxyAppPool -
processModel.identityType:SpecificUser -processModel.userName:%UserName% -processModel.password:%Password%

rem Make the 'Symbols' Virtual Directory and assign the 'SymProxy App Pool'
%windir%\system32\inetsrv\appcmd.exe add app -site.name:"Default Web Site" -path:/Symbols -
physicalpath:%VirDirectory%
%windir%\system32\inetsrv\appcmd.exe set app -app.name:"Default Web Site/Symbols" -
applicationPool:SymProxyAppPool

rem Disable 'web.config' for folders under virtual directories in the 'Default Web Site'
%windir%\system32\inetsrv\appcmd.exe set config -section:system.applicationHost/sites "/[name='Default Web
Site'].virtualDirectoryDefaults.allowSubDirConfig:false

rem Add the 'SymProxy ISAPI Filter'
%windir%\system32\inetsrv\appcmd.exe set config -section:system.webServer/isapiFilters /+"
[name='SymProxy',path='%windir%\system32\inetsrv\SymProxy.dll',enabled='True']

rem Clear the MIME Types on the 'Default Web Site'
%windir%\system32\inetsrv\appcmd.exe set config -in "Default Web Site" < staticContentClear.xml

rem Add * to the MIME Types of the 'Default Web Site'
%windir%\system32\inetsrv\appcmd.exe set config "Default Web Site" -section:staticContent /+"
[fileExtension='.*',mimeType='application/octet-stream']"
```

staticContentClear.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<appcmd>
    <CONFIG CONFIG.SECTION="system.webServer/staticContent"                path="MACHINE/WEBROOT/APPHOST">
        <system.webServer-staticContent>
            <clear />
        </system.webServer-staticContent>
    </CONFIG>
</appcmd>
```

Testing the SymProxy Installation

The system should now be ready to acquire and serve files. To test it, start by restarting the IISAdmin service by running iisreset.exe. This will reload the ISAPI filter with the current IIS and SymProxy configuration.

Configure a debugger to use this symbol path:

```
srv*\\MachineName\Symbols*https://MachineName/Symbols
```

If *MissTimeout* is enabled (it is set to 300 seconds by default), running the .reload /f command twice should result in much faster execution the second time.

To view the location of the PDBs being referenced, use the lm (list modules) command. The path to the PDBs should all begin with \\MachineName\Symbols.

If directory browsing is enabled on the web site, browse to https://MachineName/Symbols to see the files that are cached.

Open the Performance Monitor and view the Symbol Proxy counters.

Open the Event Viewer and view the Microsoft\Windows\SymProxy events.

Related topics

Installing SymProxy

Other Symbol Stores

12/5/2018 • 2 minutes to read • Edit Online

It is possible to write your own symbol store creation program, rather than using SymStore.

Since SymStore transactions are all logged in CSV-format text files, you can leverage any existing SymStore log files for use in your own database program.

If you plan to use the SymSrv program provided with Debugging Tools for Windows package, it is recommended that you use SymStore as well. Updates to these two programs will always be released together, and therefore their versions will always match.

Other Symbol Servers

12/5/2018 • 2 minutes to read • Edit Online

If you wish to use a different method for your symbol search, you can provide your own symbol server DLL rather than using SymSrv.

Setting the Symbol Path

When implementing a symbol server other than SymSrv, the debugger's symbol path is set in the same way as with SymSrv. See SymSrv for an explanation of the symbol path syntax. The only change you need to make is to replace the string **symsrv.dll** with the name of your own symbol server DLL.

If you wish, you are free to use a different syntax within the parameters to indicate the use of different technologies such as UNC paths, SQL database identifiers, or Internet specifications.

Implementing Your Own Symbol Server

The central portion of the server is the code that communicates with DbgHelp to find the symbols. Every time DbgHelp requires symbols for a newly loaded module, it calls the symbol server to locate the appropriate symbol files. The symbol server locates each file according to unique parameters such as the time stamp or image size. The server returns a validated path to the requested file. To implement this, the server must export the **SymbolServer** function.

The server should also support the **SymbolServerSetOptions** and **SymbolServerGetOptions** functions. And DbgHelp will call the **SymbolServerClose** function, if it is exported by the server. See Symbol Server API for information about where these routines are documented.

You must not change the actual symbol file name returned by your symbol server. DbgHelp stores the name of a symbol file in multiple locations. Therefore, the server must return a file of the same name as that specified when the symbol was requested. This restriction is needed to assure that the symbol names displayed during symbol loading are the ones that the programmer will recognize.

Restrictions on Multiple Symbol Servers

DbgHelp supports the use of only one symbol server at a time. Your symbol path can contain multiple instances of the same symbol server DLL, but not two different symbol server DLLs. This is not much of a restriction, since you are still free to include multiple instances of a symbol server in your symbol path, each pointing to a different symbol store. But if you want to switch between two different symbol server DLLs, you will have to change the symbol path each time.

Installing Your Symbol Server

The details of your symbol server installation will depend on your situation. You might wish to set up an installation process that copies your symbol server DLL and sets the _NT_SYMBOL_PATH environment variable automatically.

Depending on the technology used in your server, you may also need to install or access the symbol data itself.

Deferred Symbol Loading

12/5/2018 • 2 minutes to read • Edit Online

By default, symbol information is not actually loaded when the target modules are loaded. Instead, symbols are loaded by the debugger as they are needed. This is called *deferred symbol loading* or *lazy symbol loading*. When this option is enabled, the debugger loads symbols whenever it encounters an unrecognized symbol.

When the symbol path is changed, for example by using the **.sympath (Set Symbol Path)** command, all loaded modules with export symbols are lazily reloaded. Symbols of modules with full PDB symbols will be lazily reloaded if the new path no longer includes the original path that was used to load the PDB symbols. If the new path still includes the original path to the PDB symbol file, those symbols will not be lazily reloaded.

When deferred symbol loading is disabled, process startup can be much slower, because all symbols are read whenever a module is loaded.

In WinDbg, the deferred symbol loading behavior can be modified for symbols that have no module prefix by using the Resolve Unqualified Symbols option on the **Debug** menu.

You can override deferred symbol loading by using the **ld (Load Symbols)** command or the **.reload (Reload Module)** command with the **/f** option. These force the specified symbols to be loaded immediately, although the loading of other symbols is deferred.

By default, deferred symbol loading is enabled. In CDB and KD, the **-s** command-line option will turn this option off. It can also be turned off in CDB by using the *LazyLoad* variable in the tools.ini file. Once the debugger is running, this option can be turned on or off by using **.symopt+0x4** or **.symopt-0x4**, respectively.

Avoiding debugger searches for un-needed symbols

6/25/2019 • 4 minutes to read • Edit Online

Last updated:

- May 27, 2007

You arrive at an interesting breakpoint while debugging your driver, only to have the debugger pause for a very long time while it attempts to load symbols for drivers that you don't own and that don't even matter for the debugging task at hand. What's going on?

By default, symbols are loaded by the debugger as they are needed. (This is called deferred symbol loading or lazy symbol loading.) The debugger looks for symbols whenever it executes a command that calls for the display of symbols. This can happen at a breakpoint if you have set a watch variable that is not valid in the current context, such as a function parameter or local variable that doesn't exist in the current stack frame, because they become invalid when the context changes. It can also happen if you simply mistype a symbol name or execute an invalid debugger command-the debugger starts looking for a matching symbol.

Why does this sometimes take so long? That depends on whether the symbol name is qualified or unqualified. A qualified symbol name is preceded with the name of the module that contains the symbol-for example, myModule!myVar. An unqualified symbol name does not specify a module name-for example, myOtherVar.

In the case of the qualified name, the debugger looks for the symbol in the specified module and, if the module is not already loaded, loads the module (assuming the module exists and contains the symbol). This happens fairly quickly.

In the case of an unqualified name, the debugger doesn't "know" which module contains the symbol, so it must look in all of them. The debugger first checks all loaded modules for the symbol and then, if it cannot match the symbol in any loaded module, the debugger continues its search by loading all unloaded modules, starting with the downstream store and ending with the symbol server, if you're using one. Obviously, this can take a lot of time.

How to prevent automatic loading for unqualified symbols

The **SYMOPT_NO_UNQUALIFIED_LOADS** option disables or enables the debugger's automatic loading of modules when it searches for an unqualified symbol. When **SYMOPT_NO_UNQUALIFIED_LOADS** is set and the debugger attempts to match an unqualified symbol, it searches only modules that have already been loaded, and stops searching when it cannot match the symbol, instead of loading unloaded modules to continue its search. This option does not affect searching for qualified names.

SYMOPT_NO_UNQUALIFIED_LOADS is off by default. To activate this option, use the **-snul** command-line option or, while the debugger is running, use **.sympt+0x100** or **.sympt-0x100** to turn the option on or off, respectively.

To see the effect of **SYMOPT_NO_UNQUALIFIED_LOADS**, try this experiment:

1. Activate noisy symbol loading (**SYMOPT_DEBUG**) by using the **-n** command-line option or, if the debugger is already running, use **.sympt+0x80000000** or the **!sym noisy** debugger extension command. **SYMOPT_DEBUG** instructs the debugger to display information about its search for symbols, such as the name of each module as it is loaded or an error message if the debugger cannot find a file.
2. Instruct the debugger to evaluate a nonexistent symbol (for example, type **?asdasdasd**). The debugger should report numerous errors while it searches for the nonexistent symbol.
3. Activate **SYMOPT_NO_UNQUALIFIED_LOADS** by using **.sympt+0x100**.

4. Repeat step 2. The debugger should search only loaded modules for the nonexistent symbol, and it should finish the task much faster.
5. To disable **SYMOPT_DEBUG**, use **.symopt-0x80000000** or the **!sym quiet** debugger extension command.

A number of options are available to control how the debugger loads and uses symbols. For a complete list of symbol options and how to use them, see "Setting Symbol Options" in the online documentation provided with Debugging Tools for Windows. The latest release of the Debugging Tools for Windows package is available as a free download from the web, or you can install the package from the Windows DDK, Platform SDK, or Customer Support Diagnostics CD.

What should you do?

- To speed up symbol searching, use qualified names in breakpoints and debugger commands whenever possible. If you want to see a symbol from a known module, qualify it with the module name; if you don't know where the symbol is, use an unqualified name. For local variables and function arguments, use **$** as the module name (for example, *$!MyVar*).
- To diagnose the causes of slow symbol loading, activate noisy symbol loading (**SYMOPT_DEBUG**) by using the **-n** command-line option or, if the debugger is already running, by using **.symopt+0x80000000** or the **!sym noisy** debugger extension command.
- To prevent the debugger from searching for symbols in unloaded modules, activate **SYMOPT_NO_UNQUALIFIED_LOADS** by using the **-snul** command-line option or, if the debugger is already running, by using **.symopt+0x100**.
- To explicitly load the modules you need for your debugging session, use debugger commands such as **.reload** or **ld**.

Related topics

WDK and WinDbg downloads

Windows Debugging

SymStore

12/5/2018 • 2 minutes to read • Edit Online

SymStore (symstore.exe) is a tool for creating symbol stores. It is included in the Debugging Tools for Windows. For more information, see Download Debugging Tools for Windows.

SymStore stores symbols in a format that enables the debugger to look up the symbols based on the time stamp and size of the image (for a .dbg or executable file), or signature and age (for a .pdb file). The advantage of the symbol store over the traditional symbol storage format is that all symbols can be stored or referenced on the same server and retrieved by the debugger without any prior knowledge of which product contains the corresponding symbol.

Note that multiple versions of .pdb symbol files (for example, public and private versions) cannot be stored on the same server, because they each contain the same signature and age.

This section includes:

SymStore Transactions

File System References and Symbol Files

SymStore Compressed Files

Symbol Storage Format

SymStore Transactions

12/5/2018 • 4 minutes to read • Edit Online

Every call to SymStore is recorded as a transaction. There are two types of transactions: add and delete.

When the symbol store is created, a directory, called "000admin", is created under the root of the server. The 000admin directory contains one file for each transaction, as well as the log files server.txt and history.txt. The server.txt file contains a list of all transactions that are currently on the server. The history.txt file contains a chronological history of all transactions.

Each time SymStore stores or removes symbol files, a new transaction number is created. Then, a file, whose name is this transaction number, is created in 000admin. This file contains a list of all the files or pointers that have been added to the symbol store during this transaction. If a transaction is deleted, SymStore will read through its transaction file to determine which files and pointers it should delete.

The **add** and **del** options specify whether an add or delete transaction is to be performed. Including the **/p** option with an add operation specifies that a pointer is to be added; omitting the **/p** option specifies that the actual symbol file is to be added.

It is also possible to create the symbol store in two separate stages. In the first stage, you use SymStore with the **/x** option to create an index file. In the second stage, you use SymStore with the **/y** option to create the actual store of files or pointers from the information in the index file.

This can be a useful technique for a variety of reasons. For instance, this allows the symbol store to be easily recreated if the store is somehow lost, as long as the index file still exists. Or perhaps the computer containing the symbol files has a slow network connection to the computer on which the symbol store will be created. In this case, you can create the index file on the same machine as the symbol files, transfer the index file to the second machine, and then create the store on the second machine.

For a full listing of all SymStore parameters, see **SymStore Command-Line Options**.

Note SymStore does not support simultaneous transactions from multiple users. It is recommended that one user be designated "administrator" of the symbol store and be responsible for all **add** and **del** transactions.

Transaction Examples

Here are two examples of SymStore adding symbol pointers for build 2195 of Windows 2000 to \\MyDir\symsrv:

```
symstore add /r /p /f \\BuildServer\BuildShare\2195free\symbols\*.* /s \\MyDir\symsrv /t "Windows 2000" /v
"Build 2195 x86 free" /c "Sample add"
symstore add /r /p /f \\BuildServer\BuildShare\2195free\symbols\*.* /s \\MyDir\symsrv /t "Windows 2000" /v
"Build 2195 x86 checked" /c "Sample add"
```

In the following example, SymStore adds the actual symbol files for an application project in \\largeapp\appserver\bins to \\MyDir\symsrv:

```
symstore add /r /f \\largeapp\appserver\bins\*.* /s \\MyDir\symsrv /t "Large Application" /v "Build 432" /c
"Sample add"
```

Here is an example of how an index file is used. First, SymStore creates an index file based on the collection of symbol files in \\largeapp\appserver\bins\. In this case, the index file is placed on a third computer, \\hubserver\hubshare. You use the **/g** option to specify that the file prefix "\\largeapp\appserver" might change in the future:

```
symstore add /r /p /g \\largeapp\appserver /f \\largeapp\appserver\bins\*.* /x
\\hubserver\hubshare\myindex.txt
```

Now suppose you move all the symbol files off of the machine \\largeapp\appserver and put them on \\myarchive\appserver. You can then create the symbol store itself from the index file \\hubserver\hubshare\myindex.txt as follows:

```
symstore add /y \\hubserver\hubshare\myindex.txt /g \\myarchive\appserver /s \\MyDir\symsrv /p /t "Large
Application" /v "Build 432" /c "Sample Add from Index"
```

Finally, here is an example of SymStore deleting a file added by a previous transaction. See "The server.txt and history.txt Files" section below for an explanation of how to determine the transaction ID (in this case, 0000000096).

```
symstore del /i 0000000096 /s \\MyDir\symsrv
```

The server.txt and history.txt Files

When a transaction is added, several items of information are added to server.txt and history.txt for future lookup capability. The following is an example of a line in server.txt and history.txt for an add transaction:

```
0000000096,add,ptr,10/09/99,00:08:32,Windows Vista SP 1,x86 fre 1.156c-RTM-2,Added from \\mybuilds\symbols,
```

This is a comma-separated line. The fields are explained as follows:

FIELD	DESCRIPTION
0000000096	Transaction ID number, as created by SymStore.
add	Type of transaction. This field can be either **add** or **del**.
ptr	Whether files or pointers were added. This field can be either **file** or **ptr**.
10/09/99	Date when transaction occurred.
00:08:32	Time when transaction started.
Windows Vista SP 1	Product.
x86 fre	Version (optional).
Added from	Comment (optional)
Unused	(Reserved for later use.)

Here are some sample lines from the transaction file 0000000096. Each line records the directory and the location of the file or pointer that was added to the directory.

```
canon800.dbg\35d9fd51b000,\\mybuilds\symbols\sp4\dll\canon800.dbg
canonlbp.dbg\35d9fd521c000,\\mybuilds\symbols\sp4\dll\canonlbp.dbg
certadm.dbg\352bf2f48000,\\mybuilds\symbols\sp4\dll\certadm.dbg
certcli.dbg\352bf2f1b000,\\mybuilds\symbols\sp4\dll\certcli.dbg
certcrpt.dbg\352bf04911000,\\mybuilds\symbols\sp4\dll\certcrpt.dbg
certenc.dbg\352bf2f7f000,\\mybuilds\symbols\sp4\dll\certenc.dbg
```

If you use a **del** transaction to undo the original **add** transactions, these lines will be removed from server.txt, and the following line will be added to history.txt:

```
0000000105,del,0000000096
```

The fields for the delete transaction are described as follows.

FIELD	DESCRIPTION
0000000105	Transaction ID number, as created by SymStore.
del	Type of transaction. This field can be either **add** or **del**.
0000000096	Transaction that was deleted.

File System References and Symbol Files

12/5/2018 • 2 minutes to read • Edit Online

Files on disk can have both long file names and automatically generated abbreviated MS-DOS compatible 8.3 short file names. After adding a symbol file to a symbol store, it is possible that the symbols in that symbol file may not be accessible during debug if the symbol file contains any abbreviated MS-DOS 8.3 file names.

When the tools create a symbol file, the version of the file name that is recorded in the symbol file debug record depends on the tools and how they are run. If a symbol file has an abbreviated MS-DOS 8.3 file name instead of the actual file name embedded in the record, symbol loading at debug time may experience problems because the abbreviated file names vary from system to system. If this problem occurs, the contents of these symbol files may not be accessible during debug. Whenever possible, the user should refrain from using abbreviated file path names when creating symbol files. Some ways to use abbreviated file names inadvertently are to use the abbreviated file path name for a source file, an *include* directory, or an included library file.

For further information, see Matching Symbol Names.

SymStore Compressed Files

12/5/2018 • 2 minutes to read • Edit Online

SymStore can be used with compressed files in two different ways:

1. Use SymStore with the **/p** option to store pointers to the symbol files. After SymStore finishes, compress the files that the pointers refer to.
2. Use SymStore with the **/x** option to create an index file. After SymStore finishes, compress the files listed in the index file. Then, use SymStore with the **/y** option (and, if you wish, the **/p** option) to store the files or pointers to the files in the symbol store. (SymStore will not need to uncompress the files to perform this operation.)

Your symbol server will be responsible for uncompressing the files at the proper time.

If you are using SymSrv as your symbol server, any compression should be done using the compress.exe tool which is available here. Compressed files should have an underscore as the last character in their file extensions (for example, module1.pd_ or module2.db_). For details, see SymSrv.

Symbol Storage Format

12/5/2018 • 2 minutes to read • Edit Online

SymStore uses the file system itself as a database. It creates a large tree of directories, with directory names based on such things as the symbol file time stamps, signatures, age, and other data.

For example, after several different acpi.dbgs have been added to the server, the directories could look like this:

```
Directory of \\mybuilds\symsrv\acpi.dbg
10/06/1999  05:46p      <DIR>          .
10/06/1999  05:46p      <DIR>          ..
10/04/1999  01:54p      <DIR>          37cdb03962040
10/04/1999  01:49p      <DIR>          37cdb04027740
10/04/1999  12:56p      <DIR>          37e3eb1c62060
10/04/1999  12:51p      <DIR>          37e3ebcc27760
10/04/1999  12:45p      <DIR>          37ed151662060
10/04/1999  12:39p      <DIR>          37ed15dd27760
10/04/1999  11:33a      <DIR>          37f03ce962020
10/04/1999  11:21a      <DIR>          37f03cf7277c0
10/06/1999  05:38p      <DIR>          37fa7f00277e0
10/06/1999  05:46p      <DIR>          37fa7f01620a0
```

In this example, the lookup path for the acpi.dbg symbol file might look like this: \\mybuilds\symsrv\acpi.dbg\37cdb03962040.

Three files may exist inside the lookup directory:

1. acpi.dbg, if the file was stored
2. file.ptr with a path to the actual symbol file, if a pointer was stored
3. refs.ptr, which contains a list of all the current locations for acpi.dbg with this timestamp and image size that are currently added to the symbol store

Displaying the directory listing of \\mybuilds\symsrv\acpi.dbg\37cdb03962040 gives the following:

```
10/04/1999  01:54p                52 file.ptr
10/04/1999  01:54p                67 refs.ptr
```

The file file.ptr contains the text string "\\mybuilds\symbols\x86\2128.chk\symbols\sys\acpi.dbg". Since there is no file called acpi.dbg in this directory, the debugger will try to find the file at \\mybuilds\symbols\x86\2128.chk\symbols\sys\acpi.dbg.

The contents of refs.ptr are used only by SymStore, not the debugger. This file contains a record of all transactions that have taken place in this directory. A sample line from refs.ptr might be:

```
0000000026,ptr,\\mybuilds\symbols\x86\2128.chk\symbols\sys\acpi.dbg
```

This shows that a pointer to \\mybuilds\symbols\x86\2128.chk\symbols\sys\acpi.dbg was added with transaction "0000000026".

Some symbol files stay constant through various products or builds or a particular product. One example of this is the Windows 2000 file msvcrt.pdb. A directory listing of \\mybuilds\symsrv\msvcrt.pdb shows that only two versions of msvcrt.pdb have been added to the symbols server:

```
Directory of \\mybuilds\symsrv\msvcrt.pdb
10/06/1999  05:37p      <DIR>          .
10/06/1999  05:37p      <DIR>          ..
10/04/1999  11:19a      <DIR>          37a8f40e2
10/06/1999  05:37p      <DIR>          37f2c2272
```

However, a directory listing of \\mybuilds\symsrv\msvcrt.pdb\37a8f40e2 shows that refs.ptr has several pointers in it.

```
Directory of \\mybuilds\symsrv\msvcrt.pdb\37a8f40e2
10/05/1999  02:50p              54     file.ptr
10/05/1999  02:50p           2,039     refs.ptr
```

The contents of \\mybuilds\symsrv\msvcrt.pdb\37a8f40e2\refs.ptr are the following:

```
0000000001,ptr,\\mybuilds\symbols\x86\2137\symbols\dll\msvcrt.pdb
0000000002,ptr,\\mybuilds\symbols\x86\2137.chk\symbols\dll\msvcrt.pdb
0000000003,ptr,\\mybuilds\symbols\x86\2138\symbols\dll\msvcrt.pdb
0000000004,ptr,\\mybuilds\symbols\x86\2138.chk\symbols\dll\msvcrt.pdb
0000000005,ptr,\\mybuilds\symbols\x86\2139\symbols\dll\msvcrt.pdb
0000000006,ptr,\\mybuilds\symbols\x86\2139.chk\symbols\dll\msvcrt.pdb
0000000007,ptr,\\mybuilds\symbols\x86\2140\symbols\dll\msvcrt.pdb
0000000008,ptr,\\mybuilds\symbols\x86\2140.chk\symbols\dll\msvcrt.pdb
0000000009,ptr,\\mybuilds\symbols\x86\2136\symbols\dll\msvcrt.pdb
0000000010,ptr,\\mybuilds\symbols\x86\2136.chk\symbols\dll\msvcrt.pdb
0000000011,ptr,\\mybuilds\symbols\x86\2135\symbols\dll\msvcrt.pdb
0000000012,ptr,\\mybuilds\symbols\x86\2135.chk\symbols\dll\msvcrt.pdb
0000000013,ptr,\\mybuilds\symbols\x86\2134\symbols\dll\msvcrt.pdb
0000000014,ptr,\\mybuilds\symbols\x86\2134.chk\symbols\dll\msvcrt.pdb
0000000015,ptr,\\mybuilds\symbols\x86\2133\symbols\dll\msvcrt.pdb
0000000016,ptr,\\mybuilds\symbols\x86\2133.chk\symbols\dll\msvcrt.pdb
0000000017,ptr,\\mybuilds\symbols\x86\2132\symbols\dll\msvcrt.pdb
0000000018,ptr,\\mybuilds\symbols\x86\2132.chk\symbols\dll\msvcrt.pdb
0000000019,ptr,\\mybuilds\symbols\x86\2131\symbols\dll\msvcrt.pdb
0000000020,ptr,\\mybuilds\symbols\x86\2131.chk\symbols\dll\msvcrt.pdb
0000000021,ptr,\\mybuilds\symbols\x86\2130\symbols\dll\msvcrt.pdb
0000000022,ptr,\\mybuilds\symbols\x86\2130.chk\symbols\dll\msvcrt.pdb
0000000023,ptr,\\mybuilds\symbols\x86\2129\symbols\dll\msvcrt.pdb
0000000024,ptr,\\mybuilds\symbols\x86\2129.chk\symbols\dll\msvcrt.pdb
0000000025,ptr,\\mybuilds\symbols\x86\2128\symbols\dll\msvcrt.pdb
0000000026,ptr,\\mybuilds\symbols\x86\2128.chk\symbols\dll\msvcrt.pdb
0000000027,ptr,\\mybuilds\symbols\x86\2141\symbols\dll\msvcrt.pdb
0000000028,ptr,\\mybuilds\symbols\x86\2141.chk\symbols\dll\msvcrt.pdb
0000000029,ptr,\\mybuilds\symbols\x86\2142\symbols\dll\msvcrt.pdb
0000000030,ptr,\\mybuilds\symbols\x86\2142.chk\symbols\dll\msvcrt.pdb
```

This shows that the same msvcrt.pdb was used for multiple builds of symbols for Windows 2000 stored on \\mybuilds\symsrv.

Here is an example of a directory that contains a mixture of file and pointer additions:

```
Directory of E:\symsrv\dbghelp.dbg\38039ff439000
10/12/1999  01:54p         141,232     dbghelp.dbg
10/13/1999  04:57p              49     file.ptr
10/13/1999  04:57p             306     refs.ptr
```

In this case, refs.ptr has the following contents:

```
0000000043,file,e:\binaries\symbols\retail\dll\dbghelp.dbg
0000000044,file,f:\binaries\symbols\retail\dll\dbghelp.dbg
0000000045,file,g:\binaries\symbols\retail\dll\dbghelp.dbg
0000000046,ptr,\\MyDir\bin\symbols\retail\dll\dbghelp.dbg
0000000047,ptr,\\foo2\bin\symbols\retail\dll\dbghelp.dbg
```

Thus, transactions 43, 44, and 45 added the same file to the server, and transactions 46 and 47 added pointers. If transactions 43, 44, and 45 are deleted, then the file dbghelp.dbg will be deleted from the directory. The directory will then have the following contents:

```
Directory of e:\symsrv\dbghelp.dbg\38039ff439000
10/13/1999  05:01p                  49 file.ptr
10/13/1999  05:01p                 130 refs.ptr
```

Now file.ptr contains "\\foo2\bin\symbols\retail\dll\dbghelp.dbg", and refs.ptr contains

```
0000000046,ptr,\\MyDir\bin\symbols\retail\dll\dbghelp.dbg
0000000047,ptr,\\foo2\bin\symbols\retail\dll\dbghelp.dbg
```

Whenever the final entry in refs.ptr is a pointer, the file file.ptr will exist and contain the path to the associated file. Whenever the final entry in refs.ptr is a file, no file.ptr will exist in this directory. Therefore, any delete operation that removes the final entry in refs.ptr may result in file.ptr being created, deleted, or changed.

How the Debugger Recognizes Symbols

12/5/2018 • 2 minutes to read • Edit Online

This section includes:

Symbol Syntax and Symbol Matching

Symbol Options

Symbol Status Abbreviations

Symbol Syntax and Symbol Matching

12/5/2018 • 3 minutes to read • Edit Online

Symbols allow you to directly manipulate tokens that are used by the program being debugged. For example, you can set a breakpoint at the function **main** with the command **bp main**, or display the integer variable **MyInt** with the command **dd MyInt L1**.

In many cases, symbols can be used as parameters in debugger commands. This is supported for most numerical parameters, and is also supported in some text parameters. In addition to general rules for symbol syntax, there are also symbol syntax rules that apply in each of these cases.

General Symbol Syntax Rules

A symbol name consists of one or more characters, but always begins with a letter, underscore (_), question mark (**?**), or dollar sign (**$**).

A symbol name may be qualified by a module name. An exclamation mark (**!**) separates the module name from the symbol (for instance, **mymodule!main**). If no module name is used, the symbol can still be prefixed with an exclamation mark. Using an exclamation mark with no module name can be especially useful, even for local variables, to indicate to a debugger command that a parameter is a name and not a hexadecimal number. For example, the variable **fade** will be read by the **dt (Display Type)** command as an address, unless it is prefixed by an exclamation mark or the -n option is used. However, to specify that a symbol is local, precede it with a dollar sign ($) and an exclamation point (!), as in **$!lime**.

Symbol names are completely case-insensitive. This means that the presence of a **myInt** and a **MyInt** in your program will not be correctly understood by the debuggers; any command that references one of these may access the other one, regardless of how the command is capitalized.

Symbol Syntax in Numerical Expressions

The debugger understands two different kinds of expressions: Microsoft Macro Assembler (MASM) expressions and C++ expressions. As far as symbols are concerned, these two forms of syntax differ as follows:

- In MASM expressions, each symbol is interpreted as an address. Depending on what the symbol refers to, this will be the address of a global variable, local variable, function, segment, module, or any other recognized label.
- In C++ expressions, each symbol is interpreted according to its type. Depending on what the symbol refers to, it may be interpreted as an integer, a data structure, a function pointer, or any other data type. A symbol that does not correspond to a C++ data type (such as an unmodified module name) will result in a syntax error.

For an explanation of when and how to use each type of syntax, see Evaluating Expressions.

If you are using MASM expression syntax, any symbol that could be interpreted as a hexadecimal number or as a register (e.g., **BadFeed**, **ebX**) should always be prefixed by an exclamation point. This makes sure the debugger recognizes it as a symbol.

The **ss (Set Symbol Suffix)** command can be used to set the symbol suffix. This instructs the debugger to automatically append "A" or "W" to any symbol name it cannot find otherwise.

Many Win32 routines exist in both ASCII and Unicode versions. These routines often have an "A" or "W" appended to the end of their names, respectively. Using a symbol suffix will aid the debugger when searching for these symbols.

Suffix matching is not active by default.

Symbol Syntax in Text Expressions

Symbols can be used in the text parameters of some commands -- for example, **bm (Set Breakpoint)** and **x (Examine Symbols)**.

These text parameters support a variety of wildcards and specifiers. See String Wildcard Syntax for details. In addition to the standard string wildcards, a text expression used to specify a symbol can be prefixed with a leading underscore. When matching this to a symbol, the debugger will treat this as any quantity of underscores, even zero.

The symbol suffix is not used when matching symbols in text expressions.

Symbol Options

12/5/2018 • 12 minutes to read • Edit Online

A number of options are available to control how symbols are loaded and used. These options can be set in a variety of ways.

The following table lists these symbol options:

FLAG	OPTION NAME	DEFAULT IN DEBUGGER	DEFAULT IN DBH
0x1	SYMOPT_CASE_INSENSITIVE	On	On
0x2	SYMOPT_UNDNAME	On	On
0x4	SYMOPT_DEFERRED_LOADS	On	Off
0x8	SYMOPT_NO_CPP	Off	Off
0x10	SYMOPT_LOAD_LINES	Off in KD and CDB On in WinDbg	On
0x20	SYMOPT_OMAP_FIND_NEAREST	On	Off
0x40	SYMOPT_LOAD_ANYTHING	Off	Off
0x80	SYMOPT_IGNORE_CVREC	Off	Off
0x100	SYMOPT_NO_UNQUALIFIED_LOADS	Off	Off
0x200	SYMOPT_FAIL_CRITICAL_ERRORS	On	Off
0x400	SYMOPT_EXACT_SYMBOLS	Off	On
0x800	SYMOPT_ALLOW_ABSOLUTE_SYMBOLS	Off	On

FLAG	OPTION NAME	DEFAULT IN DEBUGGER	DEFAULT IN DBH
0x1000	SYMOPT_IGNORE_NT_SYMPATH	Off	Off
0x2000	SYMOPT_INCLUDE_32BIT_MODULES	Off	Off
0x4000	SYMOPT_PUBLICS_ONLY	Off	Off
0x8000	SYMOPT_NO_PUBLICS	Off	Off
0x10000	SYMOPT_AUTO_PUBLICS	On	On
0x20000	SYMOPT_NO_IMAGE_SEARCH	On	Off
0x40000	SYMOPT_SECURE	Off	Off
0x80000	SYMOPT_NO_PROMPTS	On in KD and CDB Off in WinDbg	Off
0x80000000	SYMOPT_DEBUG	Off	Off

Changing the Symbol Option Settings

The **.symopt (Set Symbol Options)** command can be used to change or display the symbol option settings. In addition, a number of command-line parameters and commands are available to change these settings; these are listed in the individual SYMOPT_*XXX* sections.

You can also control all the settings at once with the **-sflags**command-line option. This option can be followed with a decimal number, or with a hexadecimal number prefixed by **0x**. It is recommended that you use hexadecimal, since the symbol flags are aligned properly that way. Be cautious in using this method, since it sets the entire bitfield and will override all the symbol handler defaults. For example, **-sflags 0x401** will not only turn on SYMOPT_EXACT_SYMBOLS and SYMOPT_CASE_INSENSITIVE, but will also turn off all the other options that normally are on by default!

The default value for the total flag bits is 0x30237 in WinDbg, 0xB0227 in CDB and KD, and 0x10C13 in the DBH tool, when these programs are launched without any symbol-related command line options.

SYMOPT_CASE_INSENSITIVE

This symbol option causes all searches for symbol names to be case-insensitive.

This option is on by default in all debuggers. Once the debugger is running, it can be turned on or off by using **.symopt+0x1** or .symopt-0x1, respectively.

This option is on by default in DBH. Once DBH is running, it can be turned on or off by using symopt +1 or symopt -1, respectively.

SYMOPT_UNDNAME

This symbol option causes public symbol names to be undecorated when they are displayed, and causes searches for symbol names to ignore symbol decorations. Private symbol names are never decorated, regardless of whether this option is active. For information on symbol name decorations, see Public and Private Symbols.

This option is on by default in all debuggers. Once the debugger is running, it can be turned on or off by using **.sympt+0x2** or .symopt-0x2, respectively.

This option is on by default in DBH. It is turned off if the -d command-line option is used. Once DBH is running, it can be turned on or off by using symopt +2 or symopt -2, respectively.

SYMOPT_DEFERRED_LOADS

This symbol option is called *deferred symbol loading* or *lazy symbol loading*. When it is active, symbols are not actually loaded when the target modules are loaded. Instead, symbols are loaded by the debugger as they are needed. See Deferred Symbol Loading for details.

This option is on by default in all debuggers. In CDB and KD, the -s command-line option will turn this option off. It can also be turned off in CDB by using the **LazyLoad** variable in the tools.ini file. Once the debugger is running, this option can be turned on or off by using **.symopt+0x4** or .symopt-0x4, respectively.

This option is off by default in DBH. Once DBH is running, it can be turned on or off by using symopt +4 or symopt -4, respectively.

SYMOPT_NO_CPP

This symbol option turns off C++ translation. When this symbol option is set, **::** is replaced by __ in all symbols.

This option is off by default in all debuggers. It can be activated by using the -snc command-line option. Once the debugger is running, it can be turned on or off by using **.symopt+0x8** or .symopt-0x8, respectively.

This option is off by default in DBH. Once DBH is running, it can be turned on or off by using symopt +8 or symopt -8, respectively.

SYMOPT_LOAD_LINES

This symbol option allows line number information to be read from source files. This option must be on for source debugging to work correctly.

In KD and CDB, this option is off by default; in WinDbg, this option is on by default. In CDB and KD, the -lines command-line option will turn this option on. Once the debugger is running, it can be turned on or off by using **.symopt+0x10** or .symopt-0x10, respectively. It can also be toggled on and off by using the **.lines (Toggle Source Line Support)** command.

This option is on by default in DBH. Once DBH is running, it can be turned on or off by using symopt +10 or symopt -10, respectively.

SYMOPT_OMAP_FIND_NEAREST

When code has been optimized and there is no symbol at the expected location, this option causes the nearest symbol to be used instead.

This option is on by default in all debuggers. Once the debugger is running, it can be turned on or off by using **.symopt+0x20** or .symopt-0x20, respectively.

This option is on by default in DBH. Once DBH is running, it can be turned on or off by using symopt +20 or symopt -20, respectively.

SYMOPT_LOAD_ANYTHING

This symbol option reduces the pickiness of the symbol handler when it is attempting to match symbols.

This option is off by default in all debuggers. Once the debugger is running, it can be turned on or off by using

.symopt+0x40 or .symopt-0x40, respectively.

This option is off by default in DBH. Once DBH is running, it can be turned on or off by using symopt +40 or symopt -40, respectively.

SYMOPT_IGNORE_CVREC

This symbol option causes the symbol handler to ignore the CV record in the loaded image header when searching for symbols.

This option is off by default in all debuggers. It can be activated by using the -sicv command-line option. Once the debugger is running, it can be turned on or off by using **.symopt+0x80** or .symopt-0x80, respectively.

This option is off by default in DBH. Once DBH is running, it can be turned on or off by using symopt +80 or symopt -80, respectively.

SYMOPT_NO_UNQUALIFIED_LOADS

This symbol option disables the symbol handler's automatic loading of modules. When this option is set and the debugger attempts to match a symbol, it will only search modules which have already been loaded.

This option can be used as a defense against mistyping a symbol name. Normally, a mistyped symbol will cause the debugger to pause while it searches all unloaded symbol files. When this option is active, a mistyped symbol will not be found in the loaded modules, and then the search will terminate.

This option is off by default in all debuggers. It can be activated by using the -snul command-line option. Once the debugger is running, it can be turned on or off by using **.symopt+0x100** or .symopt-0x100, respectively.

This option is off by default in DBH. Once DBH is running, it can be turned on or off by using symopt +100 or symopt -100, respectively.

SYMOPT_FAIL_CRITICAL_ERRORS

This symbol option causes file access error dialog boxes to be suppressed.

If this option is off, file access errors, such as "drive not ready", encountered during symbol loading, will result in dialog boxes appearing. If this option is on, these boxes are suppressed and all access errors receive a "fail" response.

This option is on by default in all debuggers. It can be deactivated by using the -sdce command-line option. Once the debugger is running, it can be turned on or off by using **.symopt+0x200** or .symopt-0x200, respectively.

This option is off by default in DBH. Once DBH is running, it can be turned on or off by using symopt +200 or symopt -200, respectively.

SYMOPT_EXACT_SYMBOLS

This symbol option causes the debugger to perform a strict evaluation of all symbol files.

When this option is on, even the slightest discrepancy between the symbol files and the symbol handler's expectations will cause the symbols to be ignored.

This option is off by default in all debuggers. It can be activated by using the -ses command-line option. Once the debugger is running, it can be turned on or off by using **.symopt+0x400** or .symopt-0x400, respectively.

The -failinc command-line option also turns on SYMOPT_EXACT_SYMBOLS. In addition, if you are debugging a user-mode minidump or a kernel-mode minidump, -failinc will prevent the debugger from loading any modules whose images can't be mapped.

This option is on by default in DBH. Once DBH is running, it can be turned on or off by using symopt +400 or symopt -400, respectively.

SYMOPT_ALLOW_ABSOLUTE_SYMBOLS

This symbol option allows DbgHelp to read symbols that are stored at an absolute address in memory. This option is not needed in the vast majority of cases.

This option is off by default in all debuggers. Once the debugger is running, it can be turned on or off by using **.symopt+0x800** or .symopt-0x800, respectively.

This option is on by default in DBH. Once DBH is running, it can be turned on or off by using symopt +800 or symopt -800, respectively.

SYMOPT_IGNORE_NT_SYMPATH

This symbol option causes the debugger to ignore the environment variable settings for the symbol path and the executable image path.

This option is off by default in all debuggers. It can be activated by using the -sins command-line option. However, it cannot be controlled by **.symopt** once the debugger is running, because the environment variables are only read at startup.

This option is off by default in DBH, and is ignored by DBH in all cases.

SYMOPT_PUBLICS_ONLY

This symbol option causes DbgHelp to ignore private symbol data, and search only the public symbol table for symbol information. This emulates the behavior of DbgHelp before support for these types was added. see Public and Private Symbols.

This option is off by default in all debuggers. Once the debugger is running, it can be turned on or off by using **.symopt+0x4000** or .symopt-0x4000, respectively.

This option is off by default in DBH. It is turned on if the -d command-line option is used. Once DBH is running, it can be turned on or off by using symopt +4000 or symopt -4000, respectively.

SYMOPT_NO_PUBLICS

This symbol option prevents DbgHelp from searching the public symbol table. This can make symbol enumeration and symbol searches much faster. If you are concerned solely with search speed, the SYMOPT_AUTO_PUBLICS option is generally preferable to this one. For information on the public symbol table, see Public and Private Symbols.

This option is off by default in all debuggers. Once the debugger is running, it can be turned on or off by using **.symopt+0x8000** or .symopt-0x8000, respectively.

This option is off by default in DBH. Once DBH is running, it can be turned on or off by using symopt +8000 or symopt -8000, respectively.

SYMOPT_AUTO_PUBLICS

This symbol option causes DbgHelp to search the public symbol table in a .pdb file only as a last resort. If any matches are found when searching the private symbol data, the public symbols will not be searched. This improves symbol search speed.

This option is on by default in all debuggers. It can be deactivated by using the -sup command-line option. Once the debugger is running, it can be turned on or off by using **.symopt+0x10000** or .symopt-0x10000, respectively.

This option is on by default in DBH. It is turned off if the -d command-line option is used. Once DBH is running, it can be turned on or off by using symopt +10000 or symopt -10000, respectively.

SYMOPT_NO_IMAGE_SEARCH

This symbol option prevents DbgHelp from searching the disk for a copy of the image when symbols are loaded.

This option is on by default in all debuggers. Once the debugger is running, it can be turned on or off by using

.symopt+0x20000 or .symopt-0x20000, respectively.

This option is off by default in DBH. Once DBH is running, it can be turned on or off by using symopt +20000 or symopt -20000, respectively.

SYMOPT_SECURE

(Kernel mode only) This symbol option indicates whether Secure Mode is active.

Secure Mode is off by default in all debuggers. It can be activated by using the -secure command-line option. If the debugger is running, is in dormant mode, and has not established any Debugging Servers, Secure Mode can be turned on by using **.symopt+0x40000** or **.secure (Activate Secure Mode)**.

This option is off by default in DBH. Once DBH is running, it can be turned on or off by using symopt +40000 or symopt -40000, respectively.

Secure mode can never be turned off once it has been activated.

SYMOPT_NO_PROMPTS

This symbol option suppresses authentication dialog boxes from the proxy server. This may result in SymSrv being unable to access a symbol store on the internet.

For details, see Firewalls and Proxy Servers.

In KD and CDB, this option is on by default; in WinDbg, this option is off by default. Once the debugger is running, it can be turned on or off by using **.symopt+0x80000** or .symopt-0x80000, respectively, followed by the **.reload (Reload Module)** command. It can also be turned on and off by using the **!sym prompts off** and **!sym prompts** extension commands, followed by the **.reload (Reload Module)** command.

This option is off by default in DBH. Once DBH is running, it can be turned on or off by using symopt +80000 or symopt -80000, respectively.

-SYMOPT_DEBUG

This symbol option turns on *noisy symbol loading*. This instructs the debugger to display information about its search for symbols.

The name of each symbol file will be displayed as it is loaded. If the debugger cannot load a symbol file, it will display an error message. Error messages for .pdb files will be displayed in text. Error messages for .dbg files will be in the form of an error code; these codes are explained in the winerror.h file.

If an image file is loaded solely to recover symbolic header information, this will be displayed as well.

This option is off by default in all debuggers. It can be activated by using the -n command-line option. Once the debugger is running, it can be turned on or off by using **.symopt+0x80000000** or .symopt-0x80000000, respectively. It can also be turned on and off by using the **!sym noisy** and **!sym quiet** extension commands.

Note This option should not be confused with noisy *source* loading -- that is controlled by the **.srcnoisy (Noisy Source Loading)** command.

This option is off by default in DBH. It can be activated by using the -n command-line option. Once DBH is running, it can be turned on or off by using symopt +80000000 or symopt -80000000, respectively. It can also be turned on and off by using the verbose on and verbose off commands.

Symbol Status Abbreviations

3/5/2019 • 2 minutes to read • Edit Online

Symbol file types and their loading status can be determined by using the **lm (List Loaded Modules)** command, the **!lmi** extension, or WinDbg's Debug | Modules menu command.

Each of these displays information about loaded modules and their symbols.

The following abbreviations are used in the displays generated by these commands:

ABBREVIATION	MEANING
deferred	The module has been loaded, but the debugger has not attempted to load the symbols. Symbols will be loaded when needed. See Deferred Symbol Loading for details.
#	There is a mismatch between the symbol file and the executable, either in their timestamps or in their checksums.
T	The timestamp is missing, not accessible, or equal to zero.
C	The checksum is missing, not accessible, or equal to zero.
DIA	Symbol files were loaded through Debug Interface Access (DIA).
Export	No actual symbol files were found, so symbol information was extracted from the binary file's export table.
M	There is a mismatch between the symbol file and the executable, either in their timestamps or in their checksums. However, symbol files have been loaded anyway due to the symbol option settings.
PERF	This binary contains performance-optimized code. Standard address arithmetic may not produce correct results.
Stripped	Debug information was stripped from the image file.
PDB	The symbols are in .pdb format.
COFF	The symbols are in common object file format (COFF) symbol format.

Symbol Problems While Debugging

12/5/2018 • 2 minutes to read • Edit Online

Invalid or missing symbols are one of the most common causes of debugger problems. When you see some sort of problem, you need to find out if you have a symbol issue.

In some cases, the solution involves acquiring the correct symbol files. In other cases, you simply need to reconfigure the debugger to recognize symbol files you already have. But if you are not able to get the correct symbol files, you will need to work around this problem and debug the target in a more limited manner.

This section includes:

Verifying Symbols

Matching Symbol Names

Reading Symbols from Paged-Out Headers

Mapping Symbols When the PEB is Paged Out

Debugging User-Mode Processes Without Symbols

Debugging Performance-Optimized Code

Verifying Symbols

12/5/2018 • 10 minutes to read • Edit Online

Symbol problems can show up in a variety of ways. Perhaps a stack trace shows incorrect information or fails to identify the names of the functions in the stack. Or perhaps a debugger command failed to understand the name of a module, function, variable, structure, or data type.

If you suspect that the debugger is not loading symbols correctly, there are several steps you can take to investigate this problem.

First, use the **lm (List Loaded Modules)** command to display the list of loaded modules with symbol information. The most useful form of this command is the following:

```
0:000> lml
```

If you are using WinDbg, the Debug | Modules menu command will let you see this information as well.

Pay particular attention to any notes or abbreviations you may see in these displays. For an interpretation of these, see Symbol Status Abbreviations.

If you don't see the proper symbol files, the first thing to do is to check the symbol path:

```
0:000> .sympath
Current Symbol Path is: d:\MyInstallation\i386\symbols\retail
```

If your symbol path is wrong, fix it. If you are using the kernel debugger make sure your local %WINDIR% is not on your symbol path.

Then reload symbols using the **.reload (Reload Module)** command:

```
0:000> .reload ModuleName
```

If your symbol path is correct, you should activate *noisy mode* so you can see which symbol files **dbghelp** is loading. Then reload your module. See Setting Symbol Options for information about how to activate noisy mode.

Here is an example of a "noisy" reload of the Microsoft Windows symbols:

```
kd> !sym noisy
kd> .reload nt
 1: Kernel Version 2081 MP Checked
 2: Kernel base = 0x80400000 PsLoadedModuleList = 0x80506fa0
 3: DBGHELP: FindExecutableImageEx-> Looking for D:\MyInstallation\i386\ntkrnlmp.exe...mismatched timestamp
 4: DBGHELP: No image file available for ntkrnlmp.exe
 5: DBGHELP: FindDebugInfoFileEx-> Looking for
 6: d:\MyInstallation\i386\symbols\retail\symbols\exe\ntkrnlmp.dbg... no file
 7: DBGHELP: FindDebugInfoFileEx-> Looking for
 8: d:\MyInstallation\i386\symbols\retail\symbols\exe\ntkrnlmp.pdb... no file
 9: DBGHELP: FindDebugInfoFileEx-> Looking for d:\MyInstallation\i386\symbols\retail\exe\ntkrnlmp.dbg... OK
10: DBGHELP: LocatePDB-> Looking for d:\MyInstallation\i386\symbols\retail\exe\ntkrnlmp.pdb... OK
11: *** WARNING: symbols checksum and timestamp is wrong 0x0036a4ea 0x00361a83 for ntkrnlmp.exe
```

The symbol handler first looks for an image that matches the module it is trying to load (lines three and four). The image itself is not always necessary, but if an incorrect one is present, the symbol handler will often fail. These lines

show that the debugger found an image at **D:\MyInstallation\i386\ntkrnlmp.exe**, but the time-date stamp didn't match. Because the time-date stamp didn't match, the search continues. Next, the debugger looks for a .dbg file and a .pdb file that match the loaded image. These are on lines 6 through 10. Line 11 indicates that even though symbols were loaded, the time-date stamp for the image did not match (that is, the symbols were wrong).

If the symbol-search encountered a catastrophic failure, you would see a message of the form:

```
ImgHlpFindDebugInfo(00000000, module.dll, c:\MyDir;c:\SomeDir, 0823345, 0) failed
```

This could be caused by items such as file system failures, network errors, and corrupt .dbg files.

Diagnosing Symbol Loading Errors

When in noisy mode, the debugger may print out error codes when it cannot load a symbol file. The error codes for .dbg files are listed in winerror.h. The .pdb error codes come from another source and the most common errors are printed in plain English text.

Some common error codes for .dbg files from winerror.h are:

0xB
ERROR_BAD_FORMAT

0x3
ERROR_PATH_NOT_FOUND

0x35
ERROR_BAD_NETPATH

It's possible that the symbol file cannot be loaded because of a networking error. If you see ERROR_BAD_FORMAT or ERROR_BAD_NETPATH and you are loading symbols from another machine on the network, try copying the symbol file to your host computer and put its path in your symbol path. Then try to reload the symbols.

Verifying Your Search Path and Symbols

Let "c:\MyDir;c:\SomeDir" represent your symbol path. Where should you look for debug information?

In cases where the binary has been stripped of debug information, such as the free builds of Windows, first look for a .dbg file in the following locations:

```
c:\MyDir\symbols\exe\ntoskrnl.dbg
c:\SomeDir\symbols\exe\ntoskrnl.dbg
c:\MyDir\exe\ntoskrnl.dbg
c:\SomeDir\exe\ntoskrnl.dbg
c:\MyDir\ntoskrnl.dbg
c:\SomeDir\ntoskrnl.dbg
current-working-directory\ntoskrnl.dbg
```

Next, look for a .pdb file in the following locations:

```
c:\MyDir\symbols\exe\ntoskrnl.pdb
c:\MyDir\exe\ntoskrnl.pdb
c:\MyDir\ntoskrnl.pdb
c:\SomeDir\symbols\exe\ntoskrnl.pdb
c:\SomeDir\exe\ntoskrnl.pdb
c:\SomeDir\ntoskrnl.pdb
current-working-directory\ntoskrnl.pdb
```

Note that in the search for the .dbg file, the debugger interleaves searching through the MyDir and SomeDir

directories, but in the .pdb search it does not.

Windows XP and later versions of Windows do not use any .dbg symbol files. See Symbols and Symbol Files for details.

Mismatched Builds

One of the most common problems in debugging failures on a machine that is often updated is mismatched symbols from different builds. Three common causes of this problem are: pointing at symbols for the wrong build, using a privately built binary without the corresponding symbols, and using the uniprocessor hardware abstraction level (HAL) and kernel symbols on a multiprocessor machine. The first two are simply a matter of matching your binaries and symbols; the third can be corrected by renaming your hal*.dbg and ntkrnlmp.dbg to hal.dbg and ntoskrnl.dbg.

To find out what build of Windows is installed on the target computer, use the **vertarget (Show Target Computer Version)** command:

```
kd> vertarget
Windows XP Kernel Version 2505 UP Free x86 compatible
Built by: 2505.main.010626-1514
Kernel base = 0x804d0000 PsLoadedModuleList = 0x80548748
Debug session time: Mon Jul 02 14:41:11 2001
System Uptime: 0 days 0:04:53
```

Testing the Symbols

Testing the symbols is more difficult. It involves verifying a stack trace on the debugger and seeing if the debug output is correct. Here's one example to try:

```
kd> u videoprt!videoportfindadapter2
Loading symbols for 0xf2860000     videoprt.sys ->   videoprt.sys

VIDEOPRT!VideoPortFindAdapter2:
f2856f42 55              push    ebp
f2856f43 8bec            mov     ebp,esp
f2856f45 81ecb8010000    sub     esp,0x1b8
f2856f4b 8b4518          mov     eax,[ebp+0x18]
f2856f4e 53              push    ebx
f2856f4f 8365f400        and     dword ptr [ebp-0xc],0x
f2856f53 8065ff00        and     byte ptr [ebp-0x1],0x0
f2856f57 56              push    esi
```

The **u** command unassembles the videoportfindadapter string in videoprt.sys. The symbols are correct on the debugger because common stack commands like **push** and **mov** show up on the stack. Most functions begin with an add, sub, or push operation using either the base pointer (ebp) or the stack pointer (esp).

It's usually obvious when the symbols aren't working correctly. Glintmp.sys doesn't have symbols in this example because a function isn't listed next to **Glintmp**:

```
kd> kb
Loading symbols for 0xf28d0000     videoprt.sys ->   videoprt.sys
Loading symbols for 0xf9cdd000      glintmp.sys ->   glintmp.sys
*** ERROR: Symbols could not be loaded for glintmp.sys
ChildEBP RetAddr  Args to Child
f29bf1b0 8045b5fa 00000001 0000a100 00000030 ntoskrnl!RtlpBreakWithStatusInstruction
f29bf1b0 8044904e 00000001 0000a100 00000030 ntoskrnl!KeUpdateSystemTime+0x13e
f29bf234 f28d1955 f9b7d000 ffafb2dc f9b7d000 ntoskrnl!READ_REGISTER_ULONG+0x6
f29bf248 f9cde411 f9b7d000 f29bf2b0 f9ba0060 VIDEOPRT!VideoPortReadRegisterUlong+0x27
00000002 00000000 00000000 00000000 00000000 glintMP+0x1411 [No function listed.]
```

The wrong build symbols were loaded for this stack trace. Notice how there are no functions listed for the first two calls. This stack trace looks like a problem with win32k.sys drawing rectangles:

```
1: kd>
1: kd> kb                      [Local        9:50 AM]
Loading symbols for 0xf22b0000       agpcpq.sys ->   agpcpq.sys
*** WARNING: symbols checksum is wrong 0x0000735a 0x00000000 for agpcpq.sys
*** ERROR: Symbols could not be loaded for agpcpq.sys
Loading symbols for 0xa0000000       win32k.sys ->   win32k.sys
*** WARNING: symbols checksum is wrong 0x00191a41 0x001995a9 for win32k.sys
ChildEBP RetAddr  Args to Child
be682b18 f22b372b 82707128 f21c1ffc 826a70f8 agpCPQ+0x125b [No function listed.]
be682b4c a0140dd4 826a72f0 e11410a8 a0139605 agpCPQ+0x372b [No function listed.]
be682b80 a00f5646 e1145100 e1cee560 e1cee560 win32k!vPatCpyRect1_6x6+0x20b
00000001 00000000 00000000 00000000 00000000 win32k!RemoteRedrawRectangle+0x32
```

Here's the correct stack trace. The problem is really with AGP440.sys. The first item appearing on a stack trace is usually at fault. Notice that the win32k.sys rectangle error is gone:

```
1: kd> kb                      [Local        9:49 AM]
ChildEBP RetAddr  Args to Child
be682b18 f22b372b 82707128 f21c1ffc 826a70f8 agpCPQ!AgpReleaseMemory+0x88
be682b30 f20a385c 82703638 e183ec68 00000000 agpCPQ!AgpInterfaceReleaseMemory+0x8b
be682b4c a0140dd4 826a72f0 e11410a8 a0139605 VIDEOPRT!AgpReleasePhysical+0x44
be682b58 a0139605 e1cee560 e11410a8 a00e5f0a win32k!OsAGPFree+0x14
be682b64 a00e5f0a e1cee560 e11410a8 e1cee560 win32k!AGPFree+0xd
be682b80 a00f5646 e1145100 e1cee560 e1cee560 win32k!HeapVidMemFini+0x49
be682b9c a00f5c20 e1cee008 e1cee008 be682c0c win32k!vDdDisableDriver+0x3a
be682bac a00da510 e1cee008 00000000 be682c0c win32k!vDdDisableDirectDraw+0x2d
be682bc4 a00da787 00000000 e1843df8 e1843de8 win32k!PDEVOBJ__vDisableSurface+0x27
be682bec a00d59fb 00000000 e1843de8 00000000 win32k!PDEVOBJ__vUnreferencePdev+0x204
be682c04 a00d7421 e1cee008 82566a98 00000001 win32k!DrvDestroyMDEV+0x30
be682ce0 a00a9e7f e1843e10 e184a008 00000000 win32k!DrvChangeDisplaySettings+0x8b3
be682d20 a008b543 00000000 00000000 00000000 win32k!xxxUserChangeDisplaySettings+0x106
be682d48 8045d119 00000000 00000000 00000000 win32k!NtUserChangeDisplaySettings+0x48
be682d48 77e63660 00000000 00000000 00000000 ntkrnlmp!KiSystemService+0xc9
```

Useful Commands and Extensions

The following commands and extensions may be useful in tracking down symbol problems:

lm (List Loaded Modules)

Lists all modules and gives the loading status of all symbols in these modules.

!dh image-header-base

Displays header information for a loaded image beginning at *image-header-base*.

.reload /n

Reloads all kernel symbols.

.reload [image-name]

(CDB or WinDbg only) Reloads symbols for the image *image-name*. If no *image-name* is specified, reloads symbols for all images. (It is necessary to reload symbols after the symbol path has been changed.)

!sym noisy

Turns on verbose mode for symbol loads. This can be used to get information about the module loads. See Setting Symbol Options for details.

.sympath [new-symbol-path]

Sets a new symbol path, or displays the current symbol path. See Symbol Path for details.

If the kernel symbols are correct, but you aren't getting a complete stack, the following commands may also be useful:

X *!
This will list the modules which currently have symbols loaded. This is useful if the kernel symbols are correct.

.reload /user
This will attempt to reload all user-mode symbols. This is needed while performing kernel debugging if symbols were loaded while one process was running, and a break later occurred in another process. In this case, the user-mode symbols from the new process will not be loaded unless this command is executed.

X wdmaud!*start*
This will list only the symbols in the **wdmaud** module whose names contain the "start" string. This has the advantage that it forces the reloading of all the symbols in **wdmaud**, but only displays those with "start" in them. (This means a shorter listing, but since there are always some symbols with "start" in them, there will be some verification that the load has taken place.)

One other useful technique for verifying symbols is unassembling code. Most functions begin with an add, sub, or push operation using either the base pointer (**ebp**) or the stack pointer (**esp** or **sp**). Try unassembling (**U Function**) some of the functions on the stack (from offset zero) to verify the symbols.

Network and Port Problems

Problems will occur with the symbol files and while connecting to the debugger. Here are a few things to keep in mind if you encounter problems:

- Determine which COM port the debug cable is connected to on the test system.
- Check the boot.ini settings of the test system. Look for the **/debug** switch and check the baud rate and COM port settings.
- Network problems can interfere with debugging if the symbols files are accessed through the network.
- .dll and .sys files with the same name (for example – mga64.sys and mga64.dll) will confuse the debugger if they aren't separated into the proper directories of the symbol tree.
- The kernel debugger doesn't always like replacing the build symbol files with private symbol files. Double check the symbol path and do a **.reload***FileName* on the misbehaving symbol. The **!dlls** command is sometimes useful.

Questions and Misconceptions

Q: I've successfully loaded symbols, but the stack seems to be wrong. Is the debugger broken?

A: Not necessarily. The most likely cause of your problem is that you've got incorrect symbols. Go through the steps outlined in this section to determine whether you've loaded valid symbols or not. Do not assume that because some things work you have valid symbols. For example, you very well may be able to execute **dd nt!ntbuildnumber** or **u nt!KeInitializeProcess** with incorrect symbols. Verify that they are correct using the procedures outlined above.

Q: Will the debugger still work with incorrect symbols?

A: Yes and no. Often you can get away with symbols that don't strictly match. For example, symbols from a previous Windows build will often work in certain cases, but there is no rule as to when this will work and when it will not.

Q: I'm stopped in the kernel debugger and I want to view symbols for my user-mode process. Can I do it?

A: Mostly. The support for this scenario is poor because the kernel debugger doesn't keep enough information around to track the module loads for each process, but there's a reasonable workaround. To load symbols for a user-mode module, execute a **.reload -user** command. This will load the user-mode modules for the current

context.

Q: What does the following message mean?

```
*** WARNING: symbols checksum and timestamp is wrong 0x0036d6bf 0x0036ab55 for ntkrnlmp.exe
```

A: It means your symbols for ntkrnlmp.exe are wrong.

Matching Symbol Names

12/5/2018 • 2 minutes to read • Edit Online

In certain situations, the actual name of a symbol is replaced with an alternative form which can then result in symbol matching problems. This most commonly happens when changing between public and private symbols or when using MS-DOS compatibility 8.3 short names for files.

Public vs. Private Symbol Matching

Switching between public symbols and private symbols can sometimes cause symbol matching problems. Typically, a public symbol and the corresponding private symbol have the same name with different symbol decorations. But in some cases, they may have entirely different names. In such cases, you might have to explicitly reference both names. For example, you could set up two breakpoints: one on the public symbol, and a second one on the private symbol. For more details, see Public and Private Symbols.

MS-DOS Compatibility 8.3 Short Name Symbol Matching

Files that have very long names are sometimes given auto-generated MS-DOS compatibility 8.3 short names. Depending on the tools and options used for creating symbol files and for debugging, the file name stored in the image's debug record can be either the long name or one of these short names. If the short names is used, this can cause symbol matching problems because the short name assigned is system dependent.

For example, suppose there are two files, Longfilename1.pdb and Longfilename2.pdb. If they are put in the same directory one will have an MS-DOS compatibility 8.3 name of Longfi~1.pdb and the other will be Longfi~2.pdb. If they are not put in the same directory they will both be Longfi~1.pdb. Thus, if the associated .pdb files are copied carelessly, the short filenames can change, causing symbol matching problems. For more details, see File System References and Symbol Files.

Reading Symbols from Paged-Out Headers

7/16/2019 • 3 minutes to read • Edit Online

The kernel debugger must read the header for each loaded module's image in order to know which symbols correspond to that module.

If a module's header is paged out to disk, the debugger will not load symbols for this module. If this happens with a module that is essential to the debugging process, it can be a critical problem.

The following procedure can be used to solve this problem.

To acquire symbols for paged-out headers

1. Make a second copy of the kernel itself. It is probably easiest to put this on a network share.
2. Append the root directory of this share to the symbol path. See Symbol Path for the ways to change the symbol path.
3. Use the **.reload (Reload Module)** command.
4. Use the **!sym noisy** extension command to see more verbose output. If this is used, you will be able to see which symbols are loaded from the module images on the target computer, and which are loaded from the copy of the kernel modules.

This technique must be used with care, since the debugger has no way of verifying whether the file copies actually match the originals. So it is crucial that the version of Windows used on the network share matches the version used on the target computer.

This technique is only used for kernel-mode debugging. The operating system is capable of paging in any headers required during user-mode debugging (unless the disk holding the paging file is dismounted or otherwise inaccessible).

Here is an example of this technique being used:

```
kd> .reload
Connected to Windows XP 2268 x86 compatible target, ptr64 FALSE
Loading Kernel Symbols
.........Unable to read image header for dmload.sys at fe0be000 - NTSTATUS 0xC0000001
.........Unable to read image header for dmboot.sys at fda93000 - NTSTATUS 0xC0000001
.....................................Unable to read image header for fdc.sys at fdfc2000 - NTSTATUS 0xC0000001
...Unable to read image header for flpydisk.sys at fde4a000 - NTSTATUS 0xC0000001
.Unable to read image header for Fs_Rec.SYS at fe0c8000 - NTSTATUS 0xC0000001
.Unable to read image header for Null.SYS at fe2c4000 - NTSTATUS 0xC0000001
..................Unable to read image header for win32k.sys at a0000000 - NTSTATUS 0xC0000001
..Unable to read image header for dxg.sys at a0194000 - NTSTATUS 0xC0000001
.......Unable to read image header for ati2draa.dll at a01a4000 - NTSTATUS 0xC0000001
..Unable to read image header for ParVdm.SYS at fe116000 - NTSTATUS 0xC0000001
.......
Loading unloaded module list
.............
Loading User Symbols
Unable to retrieve the PEB address. This is usually caused
by being in the wrong process context or by paging
```

Notice that many images have inaccessible headers. Check the symbols from one of these files (in this example, fs_rec.sys):

```
kd> x fs_rec!*
*** ERROR: Module load completed but symbols could not be loaded for fs_rec.sys
```

These headers are apparently paged out. So you need to add the proper images to the symbol path:

```
kd> .sympath+ \\myserver\myshare\symbols\x86fre\symbols
Symbol search path is:
symsrv*symsrv.dll*c:\localcache*https://msdl.microsoft.com/download/symbols;\\myserver\myshare\symbols\x86fre\
symbols

kd> .reload
Connected to Windows XP 2268 x86 compatible target, ptr64 FALSE
Loading Kernel Symbols
.........Unable to read image header for dmload.sys at fe0be000 - NTSTATUS 0xC0000001
.........Unable to read image header for dmboot.sys at fda93000 - NTSTATUS 0xC0000001
.................................Unable to read image header for fdc.sys at fdfc2000 - NTSTATUS 0xC0000001
...Unable to read image header for flpydisk.sys at fde4a000 - NTSTATUS 0xC0000001
.Unable to read image header for Fs_Rec.SYS at fe0c8000 - NTSTATUS 0xC0000001
.Unable to read image header for Null.SYS at fe2c4000 - NTSTATUS 0xC0000001
................Unable to read image header for win32k.sys at a0000000 - NTSTATUS 0xC0000001
..Unable to read image header for dxg.sys at a0194000 - NTSTATUS 0xC0000001
.......Unable to read image header for ati2draa.dll at a01a4000 - NTSTATUS 0xC0000001
..Unable to read image header for ParVdm.SYS at fe116000 - NTSTATUS 0xC0000001
.......
Loading unloaded module list
.............
Loading User Symbols
Unable to retrieve the PEB address. This is usually caused
by being in the wrong process context or by paging
```

The same warnings have appeared, but the symbols themselves are now accessible:

```
kd> x fs_Rec!*
fe0c8358  Fs_Rec!_imp___allmul
fe0c8310  Fs_Rec!_imp__IoCreateDevice
fe0c835c  Fs_Rec!_imp___allshr
........
fe0c8360  Fs_Rec!ntoskrnl_NULL_THUNK_DATA
fe0c832c  Fs_Rec!_imp__KeSetEvent
fe0c9570  Fs_Rec!_NULL_IMPORT_DESCRIPTOR
```

Mapping Symbols When the PEB is Paged Out

12/5/2018 • 3 minutes to read • Edit Online

To load symbols, the debugger looks at the list of modules loaded by the operating system. The pointer to the user-mode module list is one of the items stored in the process environment block (PEB).

To reclaim memory, the Memory Manager may page out user-mode data to make space for other process or kernel mode components. The user-mode data that is paged out may include the PEB data structure. Without this data structure, the debugger cannot determine for which images to load symbols.

Note This affects symbol files only for the user-mode modules. Kernel-mode modules and symbols are not affected, as they are tracked in a different list.

Suppose a user-mode module is mapped into the current process and you want to fix the symbols for it. Find any address in the range of virtual addresses of the module. For example, suppose a module is mapped into a virtual address range that contains the address 7f78e9e000F. Enter the following command.

```
3: kd> !vad 7f78e9e000F 1
```

The command output displays information about the virtual address descriptor (VAD) for the module. The command output also includes a Reload command string that you can use to load the symbols for the module. The Reload command string includes the starting address (000007f7`8e9e0000) and size (32000) of the notepad module.

```
VAD @ fffffa80056fb960
...
Reload command: .reload notepad.exe=000007f7`8e9e0000,32000
```

To load the symbols, enter the command that was given in the Reload command string.

```
.reload notepad.exe=000007f7`8e9e0000,32000
```

Here is another example that uses a slightly different technique. The example demonstrates how to use the **!vad** extension to map symbols when the PEB is paged out. The basic idea is to find the starting address and size of the relevant DLL so that you can then use the **.reload** command to load the necessary symbols. Suppose that the address of the current process is 0xE0000126`01BA0AF0 and you want to fix the symbols for it. First, use the **!process** command to obtain the virtual address descriptor (VAD) root address:

```
kd> !process e000012601ba0af0 1
PROCESS e000012601ba0af0
    SessionId: 2  Cid: 0b50    Peb: 6fbfffde000  ParentCid: 0efc
    DirBase: 079e8461  ObjectTable: e000000600fbceb0  HandleCount: 360.
    Image: explorer.exe
    VadRoot e000012601a35e70 Vads 201 Clone 0 Private 917. Modified 2198. Locked 0.
...
```

Then use the **!vad** extension to list the VAD tree associated with the process. Those VADs labeled "EXECUTE_WRITECOPY" belong to code modules.

```
kd> !vad e000012601a35e70
VAD     level      start      end    commit
...
e0000126019f9790 ( 6)      3fff0    3fff7        -1 Private      READONLY
e000012601be1080 ( 7)   37d9bd30 37d9bd3e         2 Mapped  Exe  EXECUTE_WRITECOPY  <-- these are DLLs
e000012600acd970 ( 5)   37d9bec0 37d9bece         2 Mapped  Exe  EXECUTE_WRITECOPY
e000012601a5cba0 ( 7)   37d9c910 37d9c924         2 Mapped  Exe  EXECUTE_WRITECOPY
...
```

Then use the **!vad** extension again to find the starting address and size of the paged out memory which holds the DLL of interest. This confirms that you have found the correct DLL:

```
kd> !vad e000012601be1080 1

VAD @ e000012601be1080
  Start VPN:      37d9bd30  End VPN: 37d9bd3e  Control Area:  e00001260197b8d0
  First ProtoPte: e0000006013e00a0  Last PTE ffffffffffffffc  Commit Charge         2 (2.)
  Secured.Flink          0  Blink           0  Banked/Extend:        0
  File Offset   0
   ImageMap ViewShare EXECUTE_WRITECOPY

...
        File: \Windows\System32\ExplorerFrame.dll
```

The "Start VPN" field - in this case, 0x37D9BD30 - indicates the starting virtual page number. This must be converted to an actual address, by multiplying it by the page size. You can use the **? (Evaluate Expression)** command to multiply this value by 0x2000, which is the page size for the Itanium-based machine the example comes from.

```
kd> ? 37d9bd3e*2000
Evaluate expression: 7676040298496 = 000006fb`37a7c000
```

Then the size of the range can be converted to bytes:

```
kd> ? 37d9bd3e-37d9bd30+1 <--   computes the number of pages
Evaluate expression: 15 = 00000000`0000000f
kd> ? f*2000
Evaluate expression: 122880 = 00000000`0001e000
```

So ExplorerFrame.dll starts at address 0x000006Fb`37A7C000 and is 0x1E000 bytes large. You can load its symbols with:

```
kd> .reload /f ExplorerFrame.dll=6fb`37a7c000,1e000
```

Debugging User-Mode Processes Without Symbols

12/5/2018 • 2 minutes to read • Edit Online

It is important to have symbols on the faulting machine before starting the debugger for a user-mode failure. However, sometimes the debugger is started without symbols. If the problem is easily reproducible, you can just copy symbols and rerun. If, however, the problem may not occur again, some information can still be gleaned from the failure:

1. To figure out what the addresses mean, you'll need a computer which matches the one with the error. It should have the same platform (x86 or x64) and be loaded with the same version of Windows.
2. When you have the computer configured, copy the user-mode symbols and the binaries you want to debug onto the new machine.
3. Start CDB or WinDbg on the symbol-less machine.
4. If you don't know which application failed on the symbol-less machine, issue an **| (Process Status)** command. If that doesn't give you a name, break into KD on the symbol-less machine and do a **!process 0 0**, looking for the process ID given by the CDB command.
5. When you have the two debuggers set up -- one with symbols which hasn't hit the error, and one which has hit the error but is without symbols -- issue a **k (Display Stack Backtrace)** command on the symbol-less machine.
6. On the machine with symbols, issue a **u (Unassemble)** command for each address given on the symbol-less stack. This will give you the stack trace for the error on the symbol-less machine.
7. By looking at a stack trace you can see the module and function names involved in the call.

Debugging Performance-Optimized Code

12/5/2018 • 3 minutes to read • Edit Online

Microsoft has certain techniques that it uses to re-arrange compiled and linked code so that it executes with more efficiency. These techniques optimize the component for memory hierarchies, and are based on training scenarios.

The resulting optimization reduces paging (and page faults), and increases spatial locality between code and data. It addresses a key performance bottleneck that would be introduced by poor positioning of the original code. A component that has gone through this optimization may have its code or data block within a function moved to different locations of the binary.

In modules that have been optimized by these techniques, the locations of code and data blocks will often be found at memory addresses different than the locations where they would reside after normal compilation and linking. Furthermore, functions may have been split into many non-contiguous blocks, in order that the most commonly-used code paths can be located close to each other on the same pages.

Therefore, a function (or any symbol) plus an offset will not necessarily have the same meaning it would have in non-optimized code.

Debugging Performance-Optimized Code

When debugging, you can see if a module has been performance-optimized by using the **!lmi** extension command on any module for which symbols have been loaded:

```
0:000> !lmi ntdll
Loaded Module Info: [ntdll]
         Module: ntdll
   Base Address: 77f80000
     Image Name: ntdll.dll
   Machine Type: 332 (I386)
     Time Stamp: 394193d2 Fri Jun 09 18:03:14 2000
       CheckSum: 861b1
Characteristics: 230e stripped perf
Debug Data Dirs: Type Size     VA  Pointer
                 MISC  110,     0,   76c00 [Data not mapped]
     Image Type: DBG      - Image read successfully from symbol server.
                 c:\symbols\dll\ntdll.dbg
    Symbol Type: DIA PDB  - Symbols loaded successfully from symbol server.
                 c:\symbols\dll\ntdll.pdb
```

In this output, notice the term **perf** on the "Characteristics" line. This indicates that this performance optimization has been applied to ntdll.dll.

The debugger is able to understand a function or other symbol without an offset; this allows you to set breakpoints on functions or other labels without any problem. However, the output of a dissassembly operation may be confusing, because this disassembly will reflect the changes made by the optimizer.

Since the debugger will try to stay close to the original code, you might see some amusing results. The rule of thumb when working with performance-optimized codes is simply that you cannot perform reliable address arithmetic on optimized code.

Here is an example:

```
kd> bl
 0 e f8640ca6     0001 (0001) tcpip!IPTransmit
 1 e f8672660     0001 (0001) tcpip!IPFragment

kd> u f864b4cb
tcpip!IPTransmit+e48:
f864b4cb f3a4            rep     movsb
f864b4cd 8b75cc          mov     esi,[ebp-0x34]
f864b4d0 8b4d10          mov     ecx,[ebp+0x10]
f864b4d3 8b7da4          mov     edi,[ebp-0x5c]
f864b4d6 8bc6            mov     eax,esi
f864b4d8 6a10            push    0x10
f864b4da 034114          add     eax,[ecx+0x14]
f864b4dd 57              push    edi
```

You can see from the breakpoint list that the address of **IPTransmit** is 0xF8640CA6.

When you unassemble a section of code within this function at 0xF864B4CB, the output indicates that this is 0xE48 bytes past the beginning of the function. However, if you subtract the base of the function from this address, the actual offset appears to be 0xA825.

What is happening is this: The debugger is indeed showing a disassembly of the binary instructions beginning at 0xF864B4CB. But instead of computing the offset by simple subtraction, the debugger displays -- as best it can -- the offset to the function entry as it existed in the original code before the optimizations were performed. That value is 0xE48.

On the other hand, if you try to look at **IPTransmit**+0xE48, you will see this:

```
kd> u tcpip!iptransmit+e48
tcpip!ARPTransmit+d8:
f8641aee 0856ff          or      [esi-0x1],dl
f8641af1 75fc            jnz     tcpip!ARPTransmit+0xd9 (f8641aef)
f8641af3 57              push    edi
f8641af4 e828eeffff      call    tcpip!ARPSendData (f8640921)
f8641af9 5f              pop     edi
f8641afa 5e              pop     esi
f8641afb 5b              pop     ebx
f8641afc c9              leave
```

What is happening here is that the debugger recognizes the symbol **IPTransmit** as equivalent to the address 0xF8640CA6, and the command parser performs a simple addition to find that 0xF8640CA6 + 0xE48 = 0xF8641AEE. This address is then used as the argument for the **u (Unassemble)** command. But once this location is analyzed, the debugger discovers that this is not **IPTransmit** plus an offset of 0xE48. Indeed, it is not part of this function at all. Rather, it corresponds to the function **ARPTransmit** plus an offset of 0xD8.

The reason this happens is that performance optimization is not reversible through address arithmetic. While the debugger can take an address and deduce its original symbol and offset, it does not have enough information to take a symbol and offset and translate it to the correct address. Consequently, disassembly is not useful in these cases.

Offline Symbols for Windows Update

7/1/2019 • 2 minutes to read • Edit Online

This topic describes how you can work with offline symbols for Windows Update. It describes a procedure that can be used to decode Windows Update logs on machines that don't have access to the Microsoft symbol server.

If you find yourself needing to do this often, you should see if setting up a Symbol Proxy Server is viable for your networking configuration. For more information see SymProxy.

All the options below require you to have one machine that can connect to Microsoft's symbol server, and have the ability to copy files to or from the machine that has the logs. The machine that doesn't have access to the symbol server will be referred to as the *offline* machine, and the machine that does have access as the *online* machine.

We recommend using a single online machine per OS build version so the WU symbol cache will build month-by-month and contain the WU symbols from multiple update releases.

If you have access to an online machine with the same exact patch level as the offline machine, you have two options:

- Option 1: Copy the ETL event log to the online machine
- Option 2: Copy the Symbols to offline machine

Verify the online and offline PCs the same version level by running `winver` or `ver` on both machines.

```
C:\>ver

Microsoft Windows [Version 10.0.17134.167]
```

If you don't have access to an online machine with the same version, you'll need to go through some extra steps to create a SymChk manifest file, described later in this topic in Option 3: Create a SymChk manifest file.

Option 1: Copy the ETL event log to the online machine

1. Copy all the WindowsUpdate ETL files from `C:\Windows\logs\WindowsUpdate\` to your online machine.
2. On the online machine, open a PowerShell prompt and run the following Get-WindowsUpdateLog PowerShell command.

   ```
   Get-WindowsUpdateLog -ETLPath <path to ETLs>
   ```

   This will download the symbols needed for log analysis.

Option 2: Copy the symbols to the offline machine

1. On the online machine, open a PowerShell prompt and run "Get-WindowsUpdateLog". This will cache the symbols needed for log analysis.
2. Copy all the files in %temp%\WindowsUpdateLog\SymCache from the online machine to `%temp%\WindowsUpdateLog\SymCache` on the offline machine.
3. On the offline machine, open a PowerShell prompt and run "Get-WindowsUpdateLog" to analyze the logs.

Option 3: Create a SymChk manifest file

1. On the offline machine, follow steps at Using a Manifest File with SymChk to create a manifest for these files in the system32 directory:

```
storewuauth.dll
wuapi.dll
wuauclt.exe
wuaueng.dll
wuautoappupdate.dll
wuuhext.dll
wuuhmobile.dll
```

2. Copy the manifest to your online machine.
3. With the manifest file, use SymChk to download the symbols locally to your online PC.
4. Copy the folder and symbols you passed to SymChk to %temp%\WindowsUpdateLog\SymCache on your offline PC.
5. On the offline machine, open a PowerShell prompt and run "Get-WindowsUpdateLog" to analyze the logs.

See Also

Using a Symbol Server.

Symbol Path

Accessing Symbols for Debugging

Symbol Problems While Debugging

AgeStore

12/5/2018 • 2 minutes to read • Edit Online

The AgeStore tool (agestore.exe) deletes files in a directory or directory tree, based on their last access dates. This program is particularly useful for removing old files from the downstream store used by a symbol server, in order to conserve disk space.

This section includes:

Using AgeStore

AgeStore Command-Line Options

Using AgeStore

12/5/2018 • 2 minutes to read • Edit Online

AgeStore is a tool that deletes files in a directory or directory tree, based on their last access dates. Its primary use is for removing old files from the downstream store used by a symbol server or a source server, in order to conserve disk space. It can also be used as a general file deletion tool.

AgeStore can delete all files in a single directory (the *target directory*), or in all the directories within a tree (the *target tree*). The -s option indicates that an entire tree is to be targeted.

There are three ways to specify which files within the target directory or target tree are to be deleted. The agestore -date=Month-Day-Year command deletes all files that were last accessed prior to the specified date. The agestore -days=NumberOfDays command deletes all files that were last accessed more than the specified number of days ago. The agestore -size=SizeRemaining command deletes all files in the target directory or target tree, beginning with the least-recently-accessed files, until the total size of the remaining files is less than or equal to *SizeRemaining*.

For example, the following command deletes all files in C:\MyDir that were last accessed prior to January 7, 2008:

```
agestore c:\mydir -date=01-07-2008
```

The following command deletes all files in the directory tree subordinate to C:\symbols\downstreamstore that were last accessed over thirty days ago:

```
agestore c:\symbols\downstreamstore -days=30 -s
```

The following command deletes files in the directory tree subordinate to C:\symbols\downstreamstore, beginning with those accessed longest ago, until the total size of all files in this tree is less than or equal to 50,000 bytes:

```
agestore c:\symbols\downstreamstore -size=50000 -s
```

The -l option causes AgeStore to delete no files, but merely to list all the files that would be deleted without this option. Before you use any AgeStore command you should run the intended command with the -l option added, to verify that it will delete exactly those files you intend it to delete.

For the complete command line syntax, see **AgeStore Command-Line Options**.

Running AgeStore on Windows Vista and Later

Because AgeStore deletes files based on the last time that they were accessed, it can run successfully only if your file system stores Last Access Time (LAT) data. In the NTFS file system, LAT data storage can be either enabled or disabled. If it is disabled, AgeStore will not run, but will display the following error message instead:

```
Last-Access-Time support is disabled on this computer.
Please read the documentation for more details.
```

In Windows Vista and later versions of Windows, LAT data storage is disabled by default, and therefore AgeStore will not run unless you first enable this data.

In Windows Vista and later versions of Windows, you can use the FSUtil (Fsutil.exe) tool to enable the gathering of

LAT data. From a Command Prompt window, issue the following command:

```
fsutil behavior set disablelastaccess 0
```

To disable the gathering of LAT data, using the following command:

```
fsutil behavior set disablelastaccess 1
```

These changes take effect after the next restart of Windows.

The FAT32 file system always stores LAT information (although only the date, and not the time, are stored). Therefore, AgeStore works with FAT32 file systems. However, since AgeStore will not run when the NTFS LAT is disabled, you must enable NTFS LAT even if your file system is FAT32.

AgeStore Command-Line Options

12/5/2018 • 3 minutes to read • Edit Online

The AgeStore command line uses the following syntax. The parameters can be included in any order.

```
agestore [PathSpec] -date=Month-Day-Year Options

agestore [PathSpec] -days=NumberOfDays Options

agestore [PathSpec] -size=SizeRemaining Options

agestore [PathSpec] -size Options

agestore -?
```

Parameters

PathSpec
Specifies the target directory in which files are to be deleted. If the -s option is used, *PathSpec* specifies the root directory of the target tree in which files are to be deleted. *PathSpec* may be the absolute or relative path of a directory on the local computer, or a UNC path. If *PathSpec* contains spaces, it must be enclosed in quotation marks. If *PathSpec* is omitted, AgeStore uses the current working directory.

-date=*Month-Day-Year*
Specifies the cutoff date for deleting files. All files that were last accessed prior to the specified date will be deleted. The date must be specified in the format Month-Day-Year, with hyphens between the month and day and between the day and the year. Leading zeros in *Month* and *Day* are optional. The year can be specified with two digits or four. Thus you can indicate the date of January 5, 2008 as 01-05-2008 or 1-5-08.

-days=*NumberOfDays*
Specifies the cutoff date and time for deleting files. All files that were last accessed prior to the specified number of days ago will be deleted. *NumberOfDays* specifies an integer number of 24-hour days. For example, if the specifier -days=3 is used at 6:00 PM on February 17, 2008, all files last accessed prior to 6:00 PM on February 14, 2008 will be deleted.

-size=*SizeRemaining*
Specifies the total size of the files that should remain after the deletion, in bytes. When this switch is used, AgeStore deletes files in the target directory or target tree, beginning with the files accessed least recently, until the total size of the remaining files is less than or equal to *SizeRemaining*. When the **-s** option is used, AgeStore targets an entire directory tree, and *SizeRemaining* specifies the total size of files that should remain in this entire directory tree after deletion.

-size
Causes AgeStore to list the total size of all files in the target directory or target tree. No files are deleted.

Options
Any combination of the following options.

-l
Causes AgeStore not to delete any files, but merely to list all the files that would be deleted if this same command were run without the **-l** option.

-s

Causes AgeStore to treat the entire directory tree subordinate to *PathSpec* as the target. When the **-s** option is not used, the directory specified by *PathSpec* becomes the target directory in which files will be deleted. When the **-s** option is used, the directory specified by *PathSpec* and all subdirectories under it become the target tree in which files will be deleted.

-k
Causes AgeStore to keep any empty subdirectories. If this option is not used, AgeStore deletes the target directory if it is completely empty after the command runs. If the **-s** option is used without the **-k** option, all empty directories in the target directory tree will be deleted after AgeStore has completed its file deletion -- even the root directory itself, if it becomes empty. If there are directories in this tree that happen to be already empty before AgeStore runs, AgeStore deletes these directories as well. However, if the AgeStore command results in no file deletion (for example, if the **-size=***SizeRemaining* parameter specifies a size larger than the total size of all files in the target tree), empty directories are not deleted. If the **-s** option is not used, empty directories are never deleted, and the **-k** option is ignored.

-q
Quiet mode. If this option is not included, AgeStore lists all files as they are deleted.

-y
Suppresses the **(y/n)** prompt. If this option is not used, AgeStore prompts you with an "Are you sure?" prompt before deleting any files.

-?
Displays help text for the AgeStore command line.

Additional Information

For more information about the AgeStore tool, see Using AgeStore.

DBH

12/5/2018 • 2 minutes to read • Edit Online

The DBH tool (dbh.exe) is a command-line tool that displays information about the contents of a symbol file.

DBH exposes the functionality of the DbgHelp API (dbghelp.dll) through a convienient command-line interface. Therefore, its behavior may change as DbgHelp is updated. The source code for one version of DBH is available in the Windows Software Development Kit (SDK) for Windows 8.

This section includes:

Using DBH

Additional DBH Examples

DBH Command-Line Options

DBH Commands

For more information about the DbgHelp API, see the Debug Help Library documentation, which is installed as part of Debugging Tools for Windows if you perform a custom install and select the **SDK** feature and its subfeatures.

Using DBH

12/5/2018 • 3 minutes to read • Edit Online

DBH is a command-line tool that exposes many of the functions in the DbgHelp API (dbghelp.dll). It can display information about the contents of a symbol file, display specific details of the symbols in the file, and search through the file for symbols matching various criteria.

The functionality provided by DBH is similar to that provided within WinDbg, KD, and CDB by commands such as **x (Examine Symbols)**.

Running DBH in Interactive Mode

You start DBH with a simple command line, on which you specify the target module whose symbols you wish to investigate. A target module can be an EXE program or a PDB symbol file. You can also specify a process ID (PID) to investigate. See **DBH Command-Line Options** for the full syntax.

When DBH starts, it loads the symbols for the specified module, and then presents you with a prompt at which you can type a variety of commands. See DBH Commands for a list of available commands.

For example, the following sequence starts DBH by specifying the target process with process ID 4672, then executes the **enum** command at the DBH prompt to display symbols matching a specific pattern, and then executes the **q** command to quit DBH:

```
C:\> dbh -p:4672
           400000 : TimeTest
         77820000 : ntdll
         77740000 : kernel32

pid:4672 mod:TimeTest[400000]: enum TimeTest!ma*

 index            address     name
     1             42cc56 :   main
     3             415810 :   malloc
     5             415450 :   mainCRTStartup

pid:4672 mod:TimeTest[400000]: q

goodbye
```

Running DBH in Batch Mode

If you wish to run only a single DBH command, you can specify it at the end of the command line. This causes DBH to start, load the specified module, run the specified command, and then exit.

For example, the previous example could be replaced with a single command line:

```
C:\> dbh -p:4672 enum TimeTest!ma*
          400000 : TimeTest
        77820000 : ntdll
        77740000 : kernel32

index            address     name
    1             42cc56 :   main
    3             415810 :   malloc
    5             415450 :   mainCRTStartup
```

This method of running DBH is called *batch mode*, because it can be easily used in batch files. This version of the

command line can also be followed by a pipe (**|**) which redirects the DBH output to another program.

Specifying the Target

DBH can select a target in three ways: by the process ID of a running process, by the name of the executable, or by the name of the symbol file. For example, if there is exactly one instance of MyProg.exe currently running, with process ID 1234, then the following commands are almost equivalent:

```
C:\> dbh -v -p:1234
C:\> dbh -v c:\mydir\myprog.exe
C:\> dbh -v c:\mydir\myprog.pdb
```

One difference between these commands is that when you start DBH by specifying the process ID, DBH uses the actual virtual addresses being used by this process. When you start DBH by specifying the executable name or the symbol file name, DBH assumes that the module's base address is a standard value (for example, 0x01000000). You can then use the **base** command to specify the actual base address, thus shifting the addresses of all the symbols in the module.

DBH does not attach to the target process in the way that a debugger does. DBH cannot cause a process to begin or end, nor can it alter how that process runs. For DBH to attach to a process by its process ID, the target process has to be running, but once DBH has been started the target process can be terminated and DBH will continue to access its symbols.

Decorated and Undecorated Symbols

By default, DBH uses undecorated symbol names when displaying and searching for symbols. If you turn off the SYMOPT_UNDNAME symbol option, or include the -d option on the DBH command line, decorations will be included.

For information on symbol decorations, see Public and Private Symbols.

Exiting DBH

To exit DBH, use the **q** command at the DBH prompt.

Additional DBH Examples

12/5/2018 • 7 minutes to read • Edit Online

Here are additional examples of commands that can be issued at the DBH prompt.

Displaying Private Symbols and Public Symbols

If the target is a full symbol file, then each public symbol appears twice in the file: in the public symbol table, and in the private symbol data. The copy in the public symbol table often contains various decorations (prefixes and suffixes). For details, see Public and Private Symbols.

DBH can display information about this symbol from the private symbol data, from the public symbol table without decorations, and from the public symbol table with decorations. Here is an example in which all three of these are displayed, using the command **addr 414fe0** each time.

The first time this command appears in this example, DBH uses the default symbol options, so the resulting information comes from the private symbol data. Note that this information includes the address, size, and data type of the function **fgets**. Then, the command symopt +4000 is used, which turns on the SYMOPT_PUBLICS_ONLY option. This causes DBH to ignore the private symbol data, and therefore when the **addr 414fe0** command is run the second time, DBH uses the public symbol table, and no size or data type information is shown for the function **fgets**. Finally, the command symopt -2 is used, turning off the SYMOPT_UNDNAME option and causing DBH to include decorations. When the **addr 414fe0** runs this final time, it shows the decorated version of the function name, **_fgets**.

```
pid:4308 mod:TimeTest[400000]: addr 414fe0

fgets
   name : fgets
   addr :   414fe0
   size : 113
  flags : 0
   type : 7e
modbase :   400000
  value :        0
    reg : 0
  scope : SymTagNull (0)
    tag : SymTagFunction (5)
  index : 7d

pid:4308 mod:TimeTest[400000]: symopt +4000

Symbol Options: 0x10c13
Symbol Options: 0x14c13

pid:4308 mod:TimeTest[400000]: addr 414fe0

fgets
   name : fgets
   addr :   414fe0
   size : 0
  flags : 0
   type : 0
modbase :   400000
  value :        0
    reg : 0
  scope : SymTagNull (0)
    tag : SymTagPublicSymbol (a)
  index : 7f

pid:4308 mod:TimeTest[400000]: symopt -2

Symbol Options: 0x14c13
Symbol Options: 0x14c11

pid:4308 mod:TimeTest[400000]: addr 414fe0

_fgets
   name : _fgets
   addr :   414fe0
   size : 0
  flags : 0
   type : 0
modbase :   400000
  value :        0
    reg : 0
  scope : SymTagNull (0)
    tag : SymTagPublicSymbol (a)
  index : 7f
```

If the -d command-line option had been used, the results would have shown the decorated public name from the beginning.

Determining the Decorations of a Specific Symbol

DBH can determine the decorations on a specific symbol. This can be useful when used in conjunction with a program that requires symbols to be specified with their decorations, such as PDBCopy.

For example, suppose you know that the symbol file mysymbols.pdb contains a symbol whose undecorated name is **MyFunction1**. To find the decorated name, use the following procedure.

First, start DBH without the -d command-line option, and then use the symopt +4000 command so that all information comes from the public symbol table:

```
C:\> dbh c:\mydir\mysymbols.pdb

mysymbols [1000000]: symopt +4000

Symbol Options: 0x10c13
Symbol Options: 0x14c13
```

Next, use the **name** command or the **enum** command to display the address of the desired symbol:

```
mysymbols [1000000]: enum myfunction1

 index            address     name
   2ab            102cb4e :   MyFunction1
```

Now use symopt -2 to make symbol decorations visible, and then use the **addr** command with the address of this symbol:

```
mysymbols [1000000]: symopt -2

Symbol Options: 0x14c13
Symbol Options: 0x14c11

mysymbols [1000000]: addr 102cb4e

_MyFunction1@4
   name : _InterlockedIncrement@4
   addr :  102cb4e
   size : 0
  flags : 0
   type : 0
modbase :  1000000
  value :        0
    reg : 0
  scope : SymTagNull (0)
    tag : SymTagPublicSymbol (a)
  index : 2ab
```

This reveals that the decorated name of the symbol is **_MyFunction1@4**.

Decoding Symbol Decorations

The **undec** command can be used to reveal the meaning of C++ symbol decorations. In the following example, the decorations attached to ??_C@_03GGCAPAJC@Sep?$AA@ are decoded to indicate that it is a string:

```
dbh: undec ??_C@_03GGCAPAJC@Sep?$AA@

??_C@_03GGCAPAJC@Sep?$AA@ =
`string'
```

The following examples decode the decorations attached to three function names, revealing their prototypes:

```
dbh: undec ?gcontext@@3_KA

?gcontext@@3_KA =
unsigned __int64 gcontext


dbh: undec ?pathcpy@@YGXPAGPBG1K@Z

?pathcpy@@YGXPAGPBG1K@Z =
void __stdcall pathcpy(unsigned short *,unsigned short const *,unsigned short const *,unsigned long)


dbh: undec ?_set_new_handler@@YAP6AHI@ZP6AHI@Z@Z

?_set_new_handler@@YAP6AHI@ZP6AHI@Z@Z =
int (__cdecl*__cdecl _set_new_handler(int (__cdecl*)(unsigned int)))(unsigned int)
```

The **undec** command does not display information about initial underscores, the prefix **__imp_**, or trailing "**@***address*" decorations, which are commonly found attached to function names.

You can use the **undec** command with any string, not just the name of a symbol in the currently loaded module.

Sorting a List of Symbols by Address

If you simply want a list of symbols, sorted in address order, you can run DBH in batch mode and pipe the results to a **sort** command. The address values typically begin in the 18th column of each line, so the following command sorts the results by address:

```
dbh -p:4672 enum mymodule!* | sort /+18
```

Displaying Source Line Information

When you use a full symbol file, DBH can display source line information. This does not require access to any source files, since this information is stored in the symbol files themselves.

Here, the **line** command displays the hexadecimal address of the binary instructions corresponding to the specified source line, and it displays the symbols associated with that line. (In this example, there are no symbols associated with the line.)

```
dbh [1000000]: line myprogram.cpp#767

   file : e:\mydirectory\src\myprogram.cpp
   line : 767
   addr :  1006191
    key : 0000000000000000
disp : 0
```

Here, the **srclines** command displays the object files associated with the specified source line:

```
dbh [1000000]: srclines myprogram.cpp 767

0x1006191: e:\mydirectory\objchk\amd64\myprogram.obj
line 767 e:\mydirectory\src\myprogram.cpp
```

Note that the output of **srclines** is similar to that of the **ln (List Nearest Symbols)** debugger command.

Displaying a Data Type

The **type** command can be used to display information about a data type. Here it displays data about the CMDPROC type:

```
dbh [1000000]: type CMDPROC

   name : CMDPROC
   addr :        0
   size : 8
  flags : 0
   type : c
modbase :  1000000
  value :        0
    reg : 0
  scope : SymTagNull (0)
    tag : SymTagTypedef (11)
  index : c
```

The value listed after "tag" specifies the nature of this data type. In this case, **SymTagTypedef** indicates that this type was defined using a **typedef** statement.

Using Imaginary Symbols

The **add** command can add an imaginary symbol to the loaded module. The actual symbol file is not altered; only the image of that file in DBH's memory is changed.

The **add** command can be useful if you wish to temporarily override which symbols are associated with a given address range. In the following example, a portion of the address range associated with **MyModule!main** is overridden by the imaginary symbol **MyModule!magic**.

Here is how the module appears before the imaginary symbol is added. Note that the **main** function begins at 0x0042CC56, and has size 0x42B. So when the **addr** command is used with the address 0x0042CD10, it recognizes this address as lying within the boundaries of the **main** function:

```
pid:6040 mod:MyModule[400000]: enum timetest!ma*

 index            address     name
     1             42cc56 :   main
     3             415810 :   malloc
     5             415450 :   mainCRTStartup

pid:6040 mod:MyModule[400000]: addr 42cc56

main
   name : main
   addr :   42cc56
   size : 42b
  flags : 0
   type : 2
modbase :   400000
  value :        0
    reg : 0
  scope : SymTagNull (0)
    tag : SymTagFunction (5)
  index : 1

pid:6040 mod:MyModule[400000]: addr 42cd10

main+ba
   name : main
   addr :   42cc56
   size : 42b
  flags : 0
   type : 2
modbase :   400000
  value :        0
    reg : 0
  scope : SymTagNull (0)
    tag : SymTagFunction (5)
  index : 1
```

Now the symbol **magic** is added at the address 0x0042CD00, with size 0x10 bytes. When the **enum** command is used, the high bit in the index is set, showing that this is an imaginary symbol:

```
pid:6040 mod:MyModule[400000]: add magic 42cd00 10


pid:6040 mod:MyModule[400000]: enum timetest!ma*

 index            address     name
     1             42cc56 :   main
     3             415810 :   malloc
     5             415450 :   mainCRTStartup
  80000001             42cd00 :   magic
```

When the **addr** command is used, it looks for any symbols whose ranges include the specified address. Since this search begins with the specified address and runs backward, the address 0x004CD10 is now associated with **magic**. On the other hand, the address 0x004CD40 is still associated with **main**, because it lies outside the range of the **magic** symbol. Note also that the tag **SymTagCustom** indicates an imaginary symbol:

```
pid:6040 mod:MyModule[400000]: addr 42cd10

magic+10
   name : magic
   addr :   42cd00
   size : 10
  flags : 1000
   type : 0
modbase :   400000
  value :        0
    reg : 0
  scope : SymTagNull (0)
    tag : SymTagCustom (1a)
  index : 80000001

pid:6040 mod:MyModule[400000]: addr 42cd40

main+ea
   name : main
   addr :   42cc56
   size : 42b
  flags : 0
   type : 2
modbase :   400000
  value :        0
    reg : 0
  scope : SymTagNull (0)
    tag : SymTagFunction (5)
  index : 1
```

Finally, the **del** command can delete the symbol **magic**, returning all the symbols to their original ranges:

```
pid:6040 mod:MyModule[400000]: del magic


pid:6040 mod:MyModule[400000]: enum timetest!ma*

 index            address     name
     1             42cc56 :   main
     3             415810 :   malloc
     5             415450 :   mainCRTStartup
```

DBH Command-Line Options

12/5/2018 • 2 minutes to read • Edit Online

The DBH command line uses the following syntax.

```
dbh [Options] -p:PID [Command]

dbh [Options] ExecutableName [Command]

dbh [Options] SymbolFileName [Command]

dbh -?

dbh -??
```

Parameters

-p:*PID*
Specifies the process ID of the process whose symbols are to be loaded.

ExecutableName
Specifies the executable file whose symbols are to be loaded, including the file name extension (usually .exe or .sys). You should include a relative or absolute directory path; if no path is included, the current working directory is assumed. If the specified file is not found in this location, DBH searches for it using **SymLoadModuleEx**.

SymbolFileName
Specifies the symbol file whose symbols are to be loaded, including the file name extension (.pdb or .dbg). You should include a relative or absolute directory path; if no path is included, the current working directory is assumed.

Options
Any combination of the following options.

-d
Causes decorated names to be used when displaying symbols and searching for symbols. When this option is used, SYMOPT_PUBLICS_ONLY is turned on, and both SYMOPT_UNDNAME and SYMOPT_AUTO_PUBLICS are turned off. This is equivalent to issuing the command symopt +4000 followed by symopt -10002 after DBH is running.

-s:*Path*
Sets the symbol path to the specified *Path* value.

-n
Turns on *noisy symbol loading*. Additional information is displayed about the search for symbols. The name of each symbol file is displayed as it is loaded. If the debugger cannot load a symbol file, it displays an error message. Error messages for .pdb files are displayed in text. Error messages for .dbg files are in the form of an error code, explained in the winerror.h file. Not all of these messages are useful, but some of them may be helpful to analyze why a symbol file cannot be found or matched. If an image file is loaded solely to recover symbolic header information, this is displayed as well.

Command
Causes DBH to run, execute the specified *Command*, and then exit. For a list of possible commands, see DBH Commands.

-?

Displays help text for the DBH command line.

-??

Displays help text for the DBH command line, and displays a list of all DBH commands.

Additional Information

For more information about the DBH tool, see Using DBH.

DBH Commands

12/5/2018 • 8 minutes to read • Edit Online

From the DBH command line, you can use a variety of commands to analyze symbols and symbol files.

The following table lists the commands that control the DBH options and perform other basic tasks.

COMMAND	EFFECT
verbose [**on**|**off**]	Turns verbose mode on or off. With no parameter, displays the current verbose mode setting.
sympath [*Path*]	Sets the symbol search path. With no parameter, displays the current symbol search path.
symopt *Options* **symopt** +*Options* **symopt** -*Options* **symopt**	Sets the symbol options. With no **+** or **-**, the value of *Options* replaces the current symbol options. If **+** or **-** is used, *Options* specifies the options to be added or removed; there must be a space before the **+** or **-** but no space after it. With no parameter, the current symbol options are displayed. When DBH is launched, the default value of all the symbol options is 0x10C13. For a list of available options, see Setting Symbol Options.
help	Displays help text for the DBH commands.
quit	Quits the DBH program.

The following table lists the commands that load, unload, and rebase the target module. These commands cannot be used if DBH was started by specifying a process ID on the command line.

COMMAND	EFFECT
load *File*	Loads the specified module. *File* should specify the path, file name, and file name extension of either the executable file or the symbol file.
unload	Unloads the current module.
base *Address*	Sets the default base address to the specified value. All symbol addresses will be determined relative to this base address.

The following table lists the commands that search for files and display directory information.

COMMAND	EFFECT
findexe *File Path*	Locates the specified executable file in the specified path, using the **FindExecutableImage** routine.
finddbg *File Path*	Locates the specified .dbg file in the specified path. Including the .dbg extension is optional.
dir *File Path*	Locates the specified file in the specified path or in any subdirectory under this path, using the **EnumDirTree** routine.
srchtree *Path File*	Locates the specified file in the specified path or in any subdirectory under this path, using the **SearchTreeForFile** routine. This command is the same as **dir**, except that the parameters are reversed.
ffpath *File*	Finds the specified file in the current symbol path.

The following table lists the commands that parse the module list and control the default module. The default module and its base address are displayed on the DBH prompt.

COMMAND	EFFECT
mod *Address*	Changes the default module to the module with the specified base address.
refresh	Refreshes the module list.
omap	Displays the module OMAP structures.
epmod *PID*	Enumerates all the modules loaded for the specified process. *PID* specifies the process ID of the desired process.
info	Displays information about the currently loaded module.
obj *Mask*	Lists all object files associated with the default module that match the specified pattern. *Mask* may contain a variety of wildcard characters and specifiers; see String Wildcard Syntax for details.
src *Mask*	Lists all source files associated with the default module that match the specified pattern. *Mask* may contain a variety of wildcard characters and specifiers; see String Wildcard Syntax for details.

COMMAND	EFFECT
enummod	Enumerates all loaded modules. There is always at least one module, unless DBH is running without a target, in which case there are none.

The following table lists the commands that display and search for symbols.

COMMAND	EFFECT
enum *Module***!***Symbol*	Enumerates all symbols matching the specified module and symbol. *Module* specifies the module to search (without the file name extension). *Symbol* specifies a pattern that the symbol must contain. Both *Module* and *Symbol* may contain a variety of wildcard characters and specifiers; see String Wildcard Syntax for details.
enumaddr *Address*	Enumerates all symbols associated with the specified address.
addr *Address*	Displays detailed information about the symbols associated with the specified address.
name [*Module***!**]*Symbol*	Displays detailed information about the specified symbol. An optional *Module* specifier may be included. Wildcards should not be used, because if multiple symbols match the pattern, **name** only displays the first of them.
next [*Module***!**]*Symbol* **next** *Address*	Displays detailed information about the next symbol after the specified symbol or address. If a symbol is specified by name, an optional *Module* specifier may be included, but wildcards should not be used.
prev [*Module***!**]*Symbol* **prev** *Address*	Displays detailed information about the first symbol previous to the specified symbol or address. If a symbol is specified by name, an optional *Module* specifier may be included, but wildcards should not be used.
line *File***#***LineNum*	Displays the hexadecimal address of the binary instruction associated with the specified source line, and any symbols associated with this line. Also sets the current line number equal to the specified line number. *File* specifies the name of the source file, and *LineNum* specifies the line number within that file; these should be separated with a number sign (**#**).
srclines *File LineNum*	Displays the object files associated with the specified source line, and the hexadecimal address of the binary instruction associated with this line. Does not change the current line number. *File* specifies the name of the source file, and *LineNum* specifies the line number within that file; these should be separated with a space.

COMMAND	EFFECT
laddr *Address*	Displays the source file and line number corresponding to the symbol located at the specified address.
linenext	Increments the current line number, and displays information about the new line number.
lineprev	Decrements the current line number, and displays information about the new line number.
locals *Function* [*Mask*]	Displays all local variables contained within the specified function. If *Mask* is included, only those locals matching the specified pattern are displayed; see String Wildcard Syntax for details.
type *TypeName*	Displays detailed information about the specified data type. *TypeName* specifies the name of the data type (for example, WSTRING). If no type name matches this value, any matching symbol will be displayed. Unlike most DBH command parameters, *TypeName* is case-sensitive.
elines [*Source* [*Obj*]]	Enumerates all source lines matching the specified source mask and object mask. *Source* specifies the name of the source file, including the absolute path and file name extension. *Obj* specifies the name of the object file, including the relative path and file name extension. Both *Source* and *Obj* may contain a variety of wildcard characters and specifiers; see String Wildcard Syntax for details. If a parameter is omitted this is equivalent to using the asterisk () wildcard. If you do not wish to specify path information, prefix the file name with ** to indicate a wildcard path.**
index *Value*	Displays detailed information about the symbol with the specified index value.
scope *Address* **scope** [*Module***!**]*Symbol*	Displays detailed information about the parent of the specified symbol. The symbol may be specified by address or by name.
srch [**mask=***Symbol*] [**index=***Index*] [**tag=***Tag*] [**addr=***Address*] [**globals**]	Searches for all symbols that match the specified masks. *Symbol* specifies the symbol name. It should not include the module name, but it may contain wildcard characters and specifiers; see String Wildcard Syntax for details. *Index* specifies the hexadecimal address of a symbol to be used as the parent for the search. *Tag* specifies the hexadecimal symbol type classifier (**SymTag***Xxx*) value that must match the symbol. *Address* specifies the address of the symbol. If **globals** is included, only global symbols will be displayed.

COMMAND	EFFECT
uw *Address*	Displays the unwind information for the function at the specified address.
dtag	Displays all the symbol type classifier (**SymTag***Xxx*) values.
etypes	Enumerates all data types.
dump	Displays a complete list of all symbol information in the target file.

The following table lists the commands that relate to symbol servers and symbol stores.

COMMAND	EFFECT
home [*Path*]	Sets the home directory used by SymSrv and SrcSrv for the default downstream store. If the symbol path contains a reference to a symbol server that uses a default downstream store, then the **sym** subdirectory of the home directory will be used for the downstream store. With no parameter, **home** displays the current home directory.
srvpath *Path*	Tests whether the specified path is the path of a symbol store.
srvind *File*	Finds the symbol server index that corresponds to the specified file. The symbol server index is a unique value based on the contents of the file, regardless of whether it actually has been added to any symbol store. *File* should specify the file name and absolute path of the desired file.
fii *File*	Displays the symbol server indexes for the specified binary file and its associated files.
getfile *File Index*	Displays the file with the specified name and symbol server index. *File* specifies the name of the desired file; this should not include its path. *Index* specifies the symbol server index of the desired file. DBH uses the **SymFindFileInPath** routine to search the tree under the current symbol path for a file with this name and this index.
sup *Path File1 File2*	Stores a file in a symbol store, based on the values of the parameters. *Path* specifies the directory path of the symbol store. *File1* and *File2* are used to create a delta value, which is in turn used to determine the file being stored.

COMMAND	EFFECT
storeadd *File Store*	Adds the specified file to the specified symbol store. *Store* should be the root path of the symbol store.

The following table lists the DBH commands that apply to real and imaginary symbols.

COMMAND	EFFECT
undec *Name*	Reveals the meaning of the decorations attached to the specified symbol name. *Name* can be any string; it need not correspond to a currently loaded symbol. If *Name* contains C++ decorations, the meaning of these decorations is displayed.
add *Name Address Size*	Adds the specified imaginary symbol to the list of symbols loaded in DBH. *Name* specifies the name of the symbol to be added, *Address* specifies its hexadecimal address, and *Size* its hexadecimal size in bytes. This is treated like any other symbol in later DBH commands, until the DBH session is ended with **quit** or **unload**, or until the imaginary symbol is deleted with **del**. The actual target symbol file is not altered.
del *Name* **del** *Address*	Deletes an imaginary symbol previously added with the **add** command. The symbol can be specified either by name or by address. This cannot be used to delete real symbols.

PDBCopy

12/5/2018 • 2 minutes to read • Edit Online

The PDBCopy tool (pdbcopy.exe) is a command-line tool that removes private symbol information from a symbol file. It can also remove selected information from the public symbol table.

This section includes:

Using PDBCopy

Choosing Which Public Symbols to Remove

PDBCopy Command-Line Options

Using PDBCopy

10/25/2019 • 6 minutes to read • Edit Online

PDBCopy is a command-line tool that creates a stripped symbol file from a full symbol file. In other words, it takes a symbol file that contains both private symbol data and a public symbol table, and creates a copy of that file that contains only the public symbol table. Depending on which PDBCopy options are used, the stripped symbol file contains either the entire public symbol table or a specified subset of the public symbol table.

For the location of PDBCopy in the WDK, see **Installation directory** in Tools Included in Debugging Tools for Windows.

PDBCopy works with any PDB-format symbol file (with file name extension .pdb), but not with the older format (.dbg) symbol files.

For a description of public symbol tables and private symbol data, see Public and Private Symbols.

Removing Private Symbols

If you wish to create a stripped symbol file that contains all the public symbols and none of the private symbols, use PDBCopy with three parameters: the path and name of the original symbol file, the path and name of the new symbol file, and the -p option.

For example, the following command creates a new file, named publicsymbols.pdb, which contains the same public symbol table as mysymbols.pdb but contains none of the private symbol data:

pdbcopy mysymbols.pdb publicsymbols.pdb -p

If mysymbols.pdb happens to already be a stripped symbol file, the symbolic content of the new file and the old file will be identical.

After issuing this command, you should move the new file to a new location and rename it to the original name of the symbol file (in this example, mysymbols.pdb), because most debugging programs and symbol extraction programs look for symbols based on a specific file name. Alternatively, you could use the same file name for the input file and the output file on the PDBCopy command line, as long as different directories are specified:

pdbcopy c:\dir1\mysymbols.pdb c:\dir2\mysymbols.pdb -p

Note The destination file should not exist before PDBCopy is run. If a file with this name exists, various errors may occur.

Removing Private Symbols and Selected Public Symbols

If you wish to not only remove the private symbol data, but also reduce the amount of information in the public symbol table, you can use the -f option to specify a list of public symbols that are to be removed.

The following example illustrates this procedure:

1. Determine the full names, including decorations, of the symbols you wish to remove. If you are not sure of the decorated symbol names, you can use the DBH tool to determine them. See Determining the Decorations of a Specific Symbol for details. In this example, let us suppose that the decorated names of the symbols you wish to remove are **_myGlobal1** and **_myGlobal2**.

2. Create a text file containing a list of the symbols to be removed. Each line in this file should include the name of one symbol, including decorations, but not including module names. In this example, the file would contain the following two lines:

```
_myGlobal1
_myGlobal2
```

The file can be given any name you choose. Let us suppose that you name this file listfile.txt and place it in the directory C:\Temp.

3. Use the following PDBCopy command line:

```
pdbcopy OldPDB NewPDB -p -f:@TextFile
```

where *OldPDB* and *NewPDB* are the original symbol file and the new symbol file, and *TextFile* is the file created in step two. The -f option indicates that certain public symbols are to be removed, and the ampersand (@) indicates that these symbols are listed in the specified text file.

In the current example, the command would look like this:

```
pdbcopy c:\dir1\mysymbols.pdb c:\dir3\mysymbols.pdb -p -f:@c:\temp\listfile.txt
```

This creates a new symbol file, C:\dir3\mysymbols.pdb, which does not contain any private symbols and does not contain the two global variables you listed in listfile.txt.

As shown in this example, PDBCopy's -f option removes a specific list of public symbols. The ampersand (@) indicates that these symbols are listed in a text file. An alternate method is to list all the symbols on the command line, using the -f option without an ampersand. Thus the following command line is equivalent to the example in the procedure above:

pdbcopy c:\dir1\mysymbols.pdb c:\dir3\mysymbols.pdb -p -f:_myGlobal1 -f:_myGlobal2

Unless you wish to remove only one or two symbols, it is simpler to use a text file than to list them on the command line.

If you wish to remove the majority of public symbols from your .pdb file, the -F option is the easiest method. While the -f option requires you to list those public symbols you wish to remove, the -F option requires you to list those public symbols you do not wish to remove. All other public symbols (as well as all private symbols) will be removed. The -F option supports the same two syntax options as the -f option: either -F: followed by the name of a symbol to be retained, or -F:@ followed by the name of a text file that contains a list of the symbols to be retained. In either case, decorated symbol names must be used.

For example, the following command removes all private symbols and almost all public symbols, leaving only the symbols **_myFunction5** and **_myGlobal7**:

pdbcopy c:\dir1\mysymbols.pdb c:\dir3\mysymbols.pdb -p -F:_myFunction5 -F:_myGlobal7

If you combine multiple instances of the -f option on one line, all the specified symbols are removed. If you combine multiple instances of the -F option on one line, all the specified symbols are retained, and all other symbols are removed. You cannot combine -f with -F.

The -f and -F options cannot be used without the -p option, which removes all private symbol data. Even if your original file contains no private symbols, you must still include the -p option (although it has no effect in this case).

The -F option cannot be used to prevent the private symbol data from being removed. If you use this option with a symbol that is not included in the public symbol table, PDBCopy ignores it.

The mspdb*.dll File

PDBCopy must access either the Mspdb80.dll file or the Mspdb60.dll file in order to run. By default, PDBCopy uses Mspdb80.dll, which is the version used by Visual Studio .NET 2002 and later versions of Visual Studio. If your

symbols were built using Visual Studio 6.0 or an earlier version, you can specify the -vc6 command-line option so that PDBCopy uses Mspdb60.dll instead, although this is not required. PDBCopy looks for the appropriate file even if the -vc6 option is not used. You can find these files within your installation of Visual Studio, the Platform SDK, or the Windows Driver Kit (WDK).

Before running PDBCopy, make sure that the correct version of mspdb*.dll file is accessible to your computer, and make sure that its location is part of the command path. If it is not, you should use the **path** command to add this location to the command path.

The File Signature and the SymSrv Index

Each symbol file has a fixed signature that uniquely identifies it. SymSrv uses the signature to generate a unique "index value" for the file. If two files have different contents or different creation times, they will also have distinct signatures and distinct SymSrv index values.

Files created with PDBCopy are an exception to the rule of unique index values. When PDBCopy creates a new symbol file, it has the same signature and SymSrv index value as the old file. This feature allows one file to be replaced with the other without altering the behavior of symbol-related tools.

If you wish the new file to have a distinct signature and SymSrv index, use the -s option. In most cases you will not wish to use this option, since the most common use of PDBCopy is to create an altered symbol file that can replace the old file without causing a mismatch. A new signature may cause SymSrv to assign a different index value to the new file than to the old file, preventing new file from properly replacing the old one.

For the complete command line syntax, see **PDBCopy Command-Line Options**.

Choosing Which Public Symbols to Remove

12/5/2018 • 3 minutes to read • Edit Online

PDBCopy supplies the -f and -F options so that you can remove an arbitrary set of public symbols from a stripped symbol file, leaving only those symbols that your audience needs to access in order to perform their debugging.

A common use of PDBCopy is to create a special version of your symbol file for use by Microsoft in its Online Crash Analysis (OCA) program. OCA can designate certain functions as *inert*, which means that if the function is found on the stack trace it is ignored. A function would typically be declared inert if it is simply a wrapper or "pass-through" function that performs no significant computations. If such a function is found on the stack in a failure analysis, it can be assumed that this function itself was not at fault, and at most it passed on invalid or corrupt data that it received from routines earlier on the stack. By ignoring such functions, OCA can better determine the actual cause of the error or corruption.

Naturally, any function that you wish to declare "inert" needs to be included in the public symbol table of the symbol file used by OCA. However, these are not the only functions that need to be included, as the following example shows.

Suppose that you write a Windows driver and you use PDBCopy to remove all public symbols from its symbol file, except for **FunctionOne** and **FunctionSix**, two inert functions. Your expectation is that if either **FunctionOne** or **FunctionSix** are found on the stack after a crash, they will be ignored by OCA. If any other part of your driver is on the stack, Microsoft will supply you with the corresponding memory address and you can use the address to debug your driver.

However, let us suppose that your driver occupies memory in the following layout:

ADDRESS	CONTENTS OF MEMORY
0x1000	Base address of the module
0x2000	Beginning of **FunctionOne**
0x203F	End of **FunctionOne**
0x3000	Beginning of **FunctionSix**
0x305F	End of **FunctionSix**
0x7FFF	End of the module in memory

If the debugger finds an address on the stack, it selects the symbol with the next lower address. Since the public symbol table contains the address of each symbol but no size information, there is no way for the debugger to know if an address actually falls within the boundaries of any specific symbol.

Therefore, if a fault occurs at address 0x2031, the debugger run by Microsoft OCA correctly identifies the fault as lying within **FunctionOne**. Since this is an inert function, the debugger continues walking the stack to find the cause of the crash.

However, if a fault occurs at 0x2052, the debugger still matches this address to **FunctionOne**, even though it lies beyond the actual end of this function (0x203F).

Consequently, you must include in your stripped symbol file not only the functions you wish to expose, but also the symbols immediately following these functions. In this example, you would wish to expose **FunctionOne**, **FunctionTwo**, **FunctionSix**, and **FunctionSeven**:

ADDRESS	CONTENTS OF MEMORY
0x1000	Base address of the module
0x2000	Beginning of **FunctionOne**
0x203F	End of **FunctionOne**
0x2040	Beginning of **FunctionTwo**
0x3000	Beginning of **FunctionSix**
0x305F	End of **FunctionSix**
0x3060	Beginning of **FunctionSeven**
0x7FFF	End of the module in memory

If you include all four of these functions in the stripped symbol file, then the Microsoft OCA analysis will not mistakenly treat the address 0x2052 as part of **FunctionOne**. In this example it will assume that this address is part of **FunctionTwo**, but that is not important because you have not registered **FunctionTwo** with OCA as an inert function. The important thing is that the address 0x2052 is recognized as not falling within an inert function, and therefore OCA will recognize this as a meaningful fault within your driver and can inform you of the fault.

If you do not wish to publicize the names of the functions following each inert function, you can insert unimportant functions into your code following each inert function so that the names of these functions can be included in your public symbol file. Be sure to verify that these added functions do indeed follow your inert functions in your binary's address space, since some optimization routines may alter this, or even remove some functions entirely.

PDBCopy Command-Line Options

12/5/2018 • 3 minutes to read • Edit Online

The PDBCopy command line uses the following syntax. The parameters can be included in any order.

```
pdbcopy OldPDB NewPDB [Options]

pdbcopy OldPDB NewPDB -p [-f:Symbol] [-f:@TextFile] [Options]

pdbcopy OldPDB NewPDB -p [-F:Symbol] [-F:@TextFile] [Options]

pdbcopy InputPDB BackupPDB -CVE-2018-1037 [autofix|verbose]

pdbcopy /?
```

Parameters

OldPDB
Specifies the path and file name of the original symbol file to be read, including the .pdb file name extension. *OldPDB* may contain the absolute or relative path of a directory on the local computer, or a UNC path. If no path is specified, the current working directory is used. If *OldPDB* contains spaces, it must be enclosed in quotation marks.

NewPDB
Specifies the path and file name of the new symbol file to be created, including the .pdb file name extension. *NewPDB* may contain the absolute or relative path of a directory on the local computer, or a UNC path. This path must already exist; PDBCopy will not create a new directory. If no path is specified, the current working directory is used. If *NewPDB* contains spaces, you must enclose it in quotation marks. The specified file should not already exist; if it does, the new file may not be written, or may be written incorrectly.

-p
Causes PDBCopy to remove private symbol data from the new symbol file. If the old symbol file contains no private symbols, this option has no effect. If this option is omitted, PDBCopy creates a new file with identical symbol content as the original file.

-f:*Symbol*
Causes PDBCopy to remove the specified public symbol from the new symbol file. *Symbol* must specify the name of the symbol to be removed, including any symbol name decorations (for example, initial underscores), but not including the module name. This option requires the -p option. If you use multiple **-f** or **-f:@** parameters, PDBCopy removes all the specified symbols from the new symbol file.

-f:@*TextFile*
Causes PDBCopy to remove the public symbols listed in the specified text file from the new symbol file. *TextFile* specifies the file name and path (absolute or relative) of this file. This file can list the names of any number of symbols, one on each line, including any symbol name decorations (for example, initial underscores), but not including module names. This option requires the -p option.

-F:*Symbol*
Causes PDBCopy to remove all public and private symbols from the new symbol file, except for the specified public symbol. *Symbol* must specify the name of the symbol to be retained, including any symbol name decorations (for example, initial underscores), but not including the module name. This option requires the -p option. If multiple **-F** or **-F:@** parameters are used, all the specified symbols are retained in the new symbol file.

-F:@*TextFile*

Causes PDBCopy to remove all public and private symbols from the new symbol file, except for the public symbols listed in the specified text file. *TextFile* specifies the file name and path (absolute or relative) of this file. This file can list the names of any number of symbols, one on each line, including any symbol name decorations (for example, initial underscores), but not including module names. This option requires the -p option.

Options

Any combination of the following options. These options are case-sensitive.

-s

Causes the new symbol file to have a different signature than the old file. Normally you should not use the -s option, because a new signature may cause SymSrv to assign a different index value to the new file than to the old file, preventing new file from properly replacing the old one.

-vc6

Causes PDBCopy to use mspdb60.dll instead of mspdb80.dll. This option is never required, because PDBCopy automatically looks for the proper version of mspdb*.dll. By default, PDBCopy uses mspdb80.dll, which is the version used by Visual Studio .NET 2002 and later versions of Visual Studio. If your symbols were built using Visual Studio 6.0 or an earlier version, you can specify this command-line option so that PDBCopy will use mspdb60.dll instead. However, this is not required, since PDBCopy looks for the appropriate file even if this option is not used. Whichever version of mspdb*.dll you use must be in the executable path of the Command Prompt window from which you launch PDBCopy.

-CVE-2018-1037

Reports whether InputPDBFile has the issue described in CVE-2018-1037 and optionally remediates the problem. See KB# 4131751 - PDBCopy tool for more information and detailed usage information.

-?

Displays help text for the PDBCopy command line.

Additional Information

For more information about the PDBCopy tool, see Using PDBCopy.

SymChk

12/5/2018 • 2 minutes to read • Edit Online

SymChk (the Microsoft Symbol Checker tool), Symchk.exe, is a program that compares executable files to symbol files to verify that the correct symbols are available.

This section includes:

Using SymChk

SymChk Command-Line Options

Using SymChk

12/5/2018 • 4 minutes to read • Edit Online

The basic syntax for SymChk is as follows:

```
symchk [/r] FileNames /s SymbolPath
```

FileNames specifies one or more program files whose symbols are needed. If *FileNames* is a directory and the **/r** flag is used, this directory is explored recursively, and SymChk will try to find symbols for all program files in this directory tree. *SymbolPath* specifies where SymChk is to search for symbols.

There are many more command-line options. For a full listing, see SymChk Command-Line Options.

Obtaining symchk

Symchk, like other debugging tools, ship as part of the debugger. For more information, see Download Debugging Tools for Windows.

Once the debugging tools are installed, symchk is available in this directory for 64 bit Windows.

C:\Program Files (x86)\Windows Kits\10\Debuggers\x64

Example Usage

The symbol path specified can include any number of local directories, UNC directories, or symbol servers. Local directories and UNC directories are not searched recursively. Only the specified directory and a subdirectory based on the executable's extension are searched. For example, the query

```
symchk thisdriver.sys /s g:\symbols
```

will search g:\mysymbols and g:\mysymbols\sys.

You can specify a symbol server by using either of the following syntaxes as part of your symbol path:

```
srv*DownstreamStore*\\Server\Share
srv*\\Server\Share
```

This is very similar to using a symbol server in the debugger's symbol path. For details on this, see Using Symbol Servers and Symbol Stores.

If a downstream store is specified, SymChk will make copies of all valid symbol files found by the symbol server and place them in the downstream store. Only symbol files that are complete matches are copied downstream.

SymChk always searches the downstream store before querying the symbol server. Therefore you should be careful about using a downstream store when someone else is maintaining the symbol store. If you run SymChk once and it finds symbol files, it will copy those to the downstream store. If you then run SymChk again after these files have been altered or deleted on the symbol store, SymChk will not notice this fact, since it will find what it is looking for on the downstream store and look no further.

Note SymChk always uses SymSrv (Symsrv.dll) as its symbol server DLL. On the other hand, the debuggers can choose a symbol server DLL other than SymSrv if one is available. (SymSrv is the symbol server included in the Debugging Tools for Windows package.)

Using SymChk to determine whether symbols are private or public

To determine whether a symbol file is private or public, use the **/v** parameter so that SymChk displays verbose output. Suppose MyApp.exe and MyApp.pdb are in the folder c:\sym. Enter this command.

```
symchk /v c:\\sym\\MyApp.exe /s c:\\sym**
```

If MyApp.pdb contains private symbols, the output of SymChk looks like this.

```
[SYMCHK] Searching for symbols to c:\sym\MyApp.exe in path c:\sym
...
DBGHELP: MyApp - private symbols & lines
        c:\sym\MyApp.pdb
...
SYMCHK: FAILED files = 0
SYMCHK: PASSED + IGNORED files = 1
```

If MyApp.pdb contains only public symbols, the output of SymChk looks like this.

```
[SYMCHK] Searching for symbols to c:\sym\MyApp.exe in path c:\sym
...
DBGHELP: MyApp - public symbols
        c:\sym\MyApp.pdb
...
SYMCHK: FAILED files = 0
SYMCHK: PASSED + IGNORED files = 1
```

To limit your search so that it finds only public symbol files, use the **s** option with the **/s** parameter (**/ss**). The following command finds a match if MyApp.pdb contains only public symbols. It does not find a match if MyApp.pdb contains private symbols.

```
symchk /v c:\\sym\\MyApp.exe /ss c:\\sym
```

For more information, see Public and Private Symbols.

Examples

Here are some examples. The following command searches for symbols for the program Myapp.exe:

```
e:\debuggers> symchk f:\myapp.exe /s f:\symbols\applications

SYMCHK: Myapp.exe           FAILED  - Myapp.pdb is missing

SYMCHK: FAILED files = 1
SYMCHK: PASSED + IGNORED files = 0
```

You can try again with a different symbol path:

```
e:\debuggers> symchk f:\myapp.exe /s f:\symbols\newdirectory

SYMCHK: FAILED files = 0
SYMCHK: PASSED + IGNORED files = 1
```

The search was successful this time. If the verbose option is not used, SymChk will only list files for which it failed to find symbols. So in this example no files were listed. You can tell that the search succeeded because there is now one file listed in the "passed" category and none in the "failed" category.

A program file is ignored if it contains no executable code. Many resource files are of this type.

If you prefer to see the file names of all program files, you can use the **/v** option to generate verbose output:

```
e:\debuggers> symchk /v f:\myapp.exe /s f:\symbols\newdirectory

SYMCHK: MyApp.exe           PASSED

SYMCHK: FAILED files = 0
SYMCHK: PASSED + IGNORED files = 1
```

The following command searches for a huge number of Windows symbols in a symbol server. There are a great variety of possible error messages:

```
e:\debuggers> symchk /r c:\windows\system32 /s srv*\\manysymbols\windows

SYMCHK: msisam11.dll         FAILED  - MSISAM11.pdb is missing
SYMCHK: msuni11.dll          FAILED  - msuni11link.pdb is missing
SYMCHK: msdxm.ocx            FAILED  - Image is split correctly, but msdxm.dbg i
s missing
SYMCHK: expsrv.dll           FAILED  - Checksum doesn't match with expsrv.DBG
SYMCHK: imeshare.dll         FAILED  - imeshare.opt.pdb is missing
SYMCHK: ir32_32.dll          FAILED  - Built with no debugging information
SYMCHK: author.dll           FAILED  - rpctest.pdb is missing
SYMCHK: msvcrt40.dll         FAILED  - Built with no debugging information
......
SYMCHK: FAILED files = 211
SYMCHK: PASSED + IGNORED files = 4809
```

See Also

SymChk Command-Line Options

Using Symbol Servers and Symbol Stores

Using a Manifest File with SymChk

12/5/2018 • 2 minutes to read • Edit Online

In some cases, you might need to retrieve symbols for files that are on an isolated computer; that is, a computer that is either not on any network or is on a network that has no symbol store. In that situation, you can use the following procedure to retrieve symbols.

1. Run SymChk with the **/om** parameter to create a manifest file that describes the files for which you want to retrieve symbols.
2. Move the manifest file to a network that has a symbol store.
3. Run SymChk with the **/im** parameter to retrieve symbols for the files described in the manifest file.
4. Move the symbol files back to the isolated computer.

Example

Suppose yourApp.exe is running on an isolated computer. The following command creates a manifest file that describes all the symbols needed to debug the yourApp.exe pocess.

```
C:\>SymChk /om c:\Manifest\man.txt /ie yourApp.exe

SYMCHK: FAILED files = 0
SYMCHK: PASSED + IGNORED files = 28
```

Now assume you have moved the manifest file to a different computer that is on a network that has access to a symbol store. The following command retrieves the symbols described in the manifest file and places them in the mySymbols folder.

```
C:\>SymChk /im c:\FolderOnOtherComputer\man.txt /s srv*c:\mysymbols*\\aServer\symbols

SYMCHK: myApp.exe             ERROR - Unable to download file. Error reported was 2
. . .
SYMCHK: FAILED files = 28
SYMCHK: PASSED + IGNORED files = 28
```

Now you can move the symbols to the isolated computer and use them for debugging.

SymChk Command-Line Options

9/25/2019 • 8 minutes to read • Edit Online

SymChk uses the following syntax:

```
symchk [/r] [/v | /q ] FileNames /s[Opts] SymbolPath Options

symchk [/r] [/v | /q ] /ie ExeFile /s[Opts] SymbolPath Options

symchk [/r] [/v | /q ] /id DumpFile /s[Opts] SymbolPath Options

symchk [/r] [/v | /q ] /ih HotFixFile /s[Opts] SymbolPath Options

symchk [/r] [/v | /q ] /ip ProcessID /s[Opts] SymbolPath Options

symchk [/r] [/v | /q ] /it TextFileList /s[Opts] SymbolPath Options

symchk [/r] [/v | /q ] /om Manifest FileNames

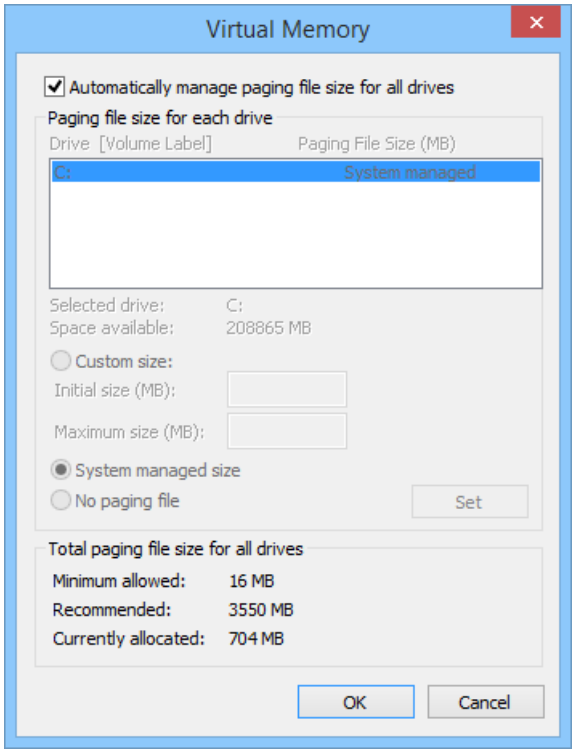
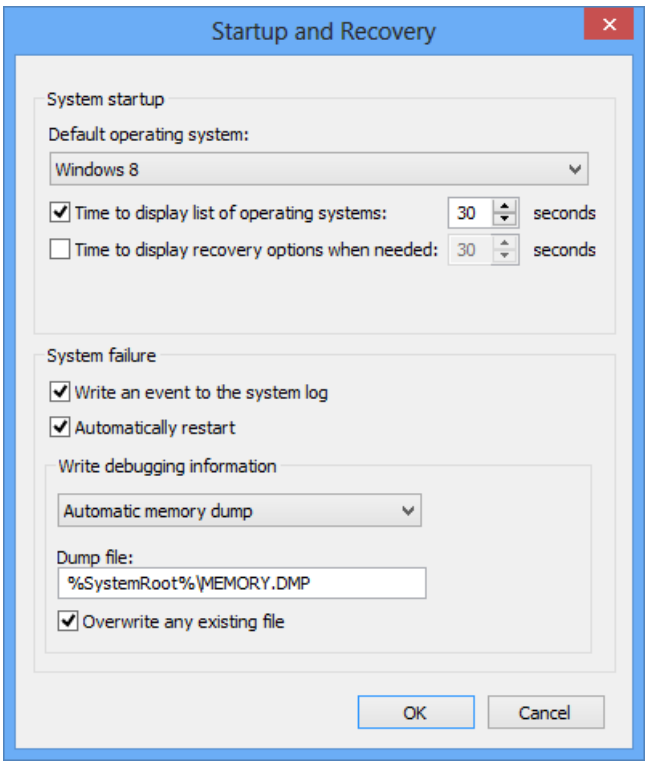
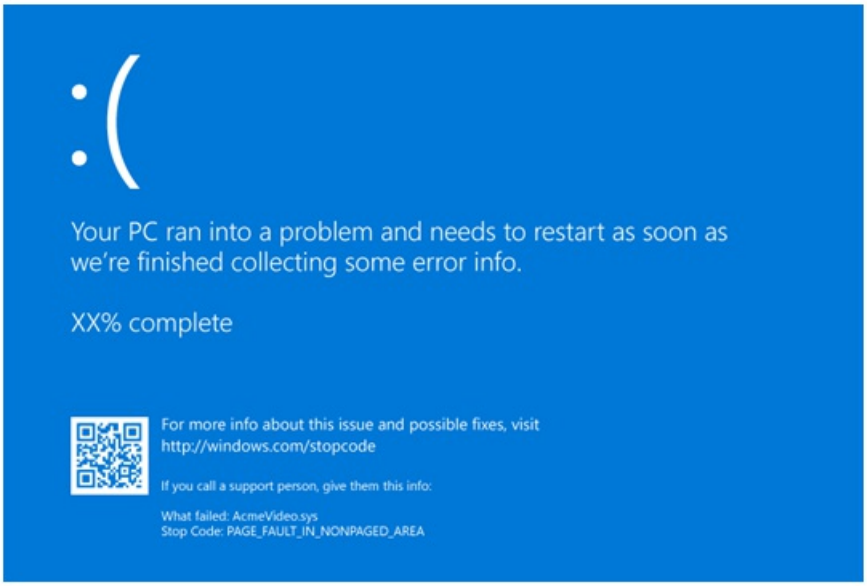
symchk [/v | /q ] /im ManifestList /s[Opts] SymbolPath Options

symchk [/v | /q ] /om Manifest /ie ExeFile

symchk [/v | /q ] /om Manifest /id DumpFile

symchk [/v | /q ] /om Manifest /ih HotFixFile

symchk [/v | /q ] /om Manifest /ip ProcessFile

symchk [/v | /q ] /om Manifest /it TextFileList
```

Parameters

/r
If *Files* specifies a directory, the **/r** option causes SymChk to recursively search all subdirectories under this directory for program files.

/v
Displays verbose information. This includes the file name of every program file whose symbols were investigated and whether it passed, failed, or was ignored.

/q
Enables quiet mode. All output will be suppressed (unless the **/ot** option is included).

FileNames
Specifies the program files whose symbols are to be checked. Absolute paths, relative paths, and UNC paths are permitted. An asterisk (*****) wildcard is permitted. If *FileNames ends in a slash, it is taken to be a directory name, and all files within that directory are checked. If *FileNames* contains spaces, it must be enclosed in quotation marks.

/ie *ExeFile*
Specifies the name of a program that is currently executing. The symbols for this program will be checked. *ExeFile* must include the name of the file and file extension (usually .exe), but no path information. If there are two different executables with the same name, this option is not recommended. *ExeFile* can specify any program, including a kernel-mode driver. If *ExeFile* is a single asterisk (*****), SymChk will check the symbols for all running

processes, including drivers.

/id *DumpFile*
Specifies a memory dump file. The symbols for this dump file will be checked.

/ih *HotFixFile*
Specifies a self-extracting Hotfix CAB file.

/ip *ProcessID*
Specifies the process ID of a program that is currently executing. The symbols for this program will be checked. *ProcessID* must be specified as a decimal number. There are two special wildcards supported:

- If *ProcessID* is zero (**0**), SymChk will check the symbols for all running drivers.
- If *ProcessID* is a single asterisk (*****), SymChk will check the symbols for all running processes, including drivers.

/it *TextFileList*
Specifies a text file that contains a list of program files. The symbols for all these programs will be checked. *TextFileList* must specify exactly one file (by relative, absolute, or UNC path, but with no wildcards); if it contains spaces it should be enclosed in quotation marks. Within this file, each line indicates a program file (by relative, absolute, or UNC paths), and an asterisk wildcard (*****) is permitted. However, any line using this wildcard must use a relative path.

If a line in this file contains spaces, it should be enclosed in quotation marks. A semicolon within this file is a comment character -- everything between a semicolon and the end of the line will be ignored.

/im *ManifestList*
Specifies that the input to the command is a manifest file previously created by using the **/om** parameter. The manifest file contains information about the files for which symbols are retrieved. For more information about using a manifest file, see Using a Manifest File with SymChk.

/om *Manifest*
Specifies that a manifest file is created. The manifest file contains information about a set of files for which symbols will be retrieved, by using the **/im** parameter, at a later time.

/s[*Opts*] *SymbolPath*
Specifies the directories containing symbols. Absolute paths, relative paths, and UNC paths are permitted. Any number of directories can be specified -- multiple directories should be separated with semicolons. If *SymbolPath* contains spaces, it must be enclosed in quotation marks. If you wish to specify a symbol server within this path, you should use one of the following syntaxes:

```
srv*DownstreamStore*\\Server\Share
srv*\\Server\Share
```

It is not recommended that you omit the **/s**[*Opts*] *SymbolPath* parameter, but if it is omitted, SymChk will point to the public symbol store by using the following default path:

```
srv*%SystemRoot%\symbols*https://msdl.microsoft.com/download/symbols
```

Any number of the following options can follow **/s**. There can be no space between the **/s** and these options:

e
SymChk will check each path individually instead of checking all paths at once.

u
Downstream stores will be updated. If the symbol path includes a downstream store, the symbol store will be

searched for the symbol files. Only symbol stores that are being checked by SymChk will be updated.

p
Force checking for private symbols. Public symbols will be treated as not matching. The **p** option implies **e** and **u**, and cannot be used with **s**.

s
Force checking for public (split) symbols. Private symbols will be treated as not matching. The **s** option implies **e** and **u**, and cannot be used with **p**.

r
Expand all non-symbol server elements in the specified path in order to do a deep search of the path. NOTE: This option may produce matches that will not occur inside the debugger since it modifies the symbol path specified.

Options
The available options are divided into several classes. Each class of options controls a different set of features.

Output options. Any number of the following options can be specified. These options can be abbreviated by using **/o** only once -- for example, **/oi /oe** can be written as **/oie**.

OPTION	EFFECT
/oe	Output will include individual errors. This option is only useful if **/q** is used, because individual errors are automatically displayed if quiet mode hasn't been activated.
/op	Output will list each file that passes. (By default, SymChk only displays files that fail testing.)
/oi	Output will list each file that was ignored. (By default, SymChk only displays files that fail testing.)
/od	Output will include full details. Same as **/oe /op /oi**.
/ot	Output will include result totals. This option is only useful if **/q** is used, because these totals are automatically displayed if quiet mode hasn't been activated.
/ob	The full path for binaries will be included in all output messages.
/os	The full path for symbols will be included in all output messages.
/oc *Dir*	SymChk will create a traditional symbol tree in the directory *Dir* that contains a list of all the symbol files checked.
/ov	SymChk will print version information for checked binaries as well.

OPTION	EFFECT
/ol *File*	In addition to the messages sent to standard out, write a file that contains a comma separated list of all the binaries and their symbols that pass symbol checking.

DBG file options. These options control how SymChk checks *.dbg* symbol files. Only one of the following options can be specified.

OPTION	EFFECT
/ds	SymChk will verify that .dbg information was stripped from the executable and only appears in the .dbg file, and that the executable points to the .dbg file. If the program was built without .dbg symbol files, this option has no effect. This is the default.
/de	SymChk will verify that .dbg information was not stripped from the executable and that the executable does not point to a .dbg file. If the program was built without .dbg symbol files, this option has no effect.
/dn	SymChk will verify that .dbg information is not present in the image, and that the image does not point to a .dbg file.

PDB file options. These options control how SymChk checks .pdb symbol files. Only one of the following options can be specified.

OPTION	EFFECT
/pf	SymChk performs no checking on the contents of the .pdb file -- it just verifies that the files exist and match the binary. This is the default.
/ps	SymChk will verify that the .pdb files have been stripped of source line, data type, and global information.
/pt	SymChk will verify that the .pdb files contain data type information.

Filtering options. These options control how module filtering is performed when SymChk is checking processes or dump files. Only one of the following options can be specified.

OPTION	EFFECT
/fm *Module*	SymChk will only check dump files or processes associated with the specified module. *Module* must include the full filename, but must not include any part of the directory path.

Symbol checking options. Any number of the following options can be specified.

OPTION	EFFECT
/cs	SymChk won't verify that CodeView data is present. (By default, the presence of CodeView data is verified.)
/cc	When SymChk is checking a hotfix CAB file, it will not look for symbols inside the cab. (By default, SymChk will look for symbols in the cab as well as in the provided symbol path.)
/ea *File*	SymChk won't verify symbols for the programs listed in the specified file. This allows you to veto certain programs that would otherwise be verified. *File* must specify exactly one file (by relative, absolute, or UNC path, but without wildcards); if it contains spaces it should be enclosed in quotation marks. Within *File*, each line indicates a program file (by relative, absolute, or UNC paths); no wildcards are permitted. If a line in this file contains spaces it should be enclosed in quotation marks. A semicolon within this file is a comment character -- everything between a semicolon and the end of the line will be ignored. If a symbol server is being used, symbols for these programs will not be copied to the downstream store.
/ee *File*	Error messages for those programs listed in the specified file are suppressed. "Success" and "ignore" messages will appear as usual, and symbol files will be copied to the downstream store as usual. The format of *File* and the format of its contents are the same as that for **/ea** *File*.

Additional Information

For more information about SymChk, see Using SymChk.

Analyze crash dump files by using WinDbg

7/18/2019 • 2 minutes to read • Edit Online

You can analyze crash dump files by using WinDbg and other Windows debuggers.

> **NOTE**
> This content is for developers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

This section includes:

Kernel-mode dump files

User-mode dump files

Extracting information from a dump file

CAB files that contain paging files along with a memory dump

Debugging OCA minidump files

Kernel-Mode Dump Files

12/5/2018 • 2 minutes to read • Edit Online

When a kernel-mode error occurs, the default behavior of Microsoft Windows is to display the blue screen with bug check data.

However, there are several alternative behaviors that can be selected:

- A kernel debugger (such as WinDbg or KD) can be contacted.
- A memory dump file can be written.
- The system can automatically reboot.
- A memory dump file can be written, and the system can automatically reboot afterwards.

This section covers how to create and analyze a kernel-mode memory dump file. There are three different varieties of crash dump files. However, it should be remembered that no dump file can ever be as useful and versatile as a live kernel debugger attached to the system that has failed.

This section includes:

Varieties of Kernel-Mode Dump Files

Creating a Kernel-Mode Dump File

Analyzing a Kernel-Mode Dump File

Varieties of Kernel-Mode Dump Files

12/5/2018 • 2 minutes to read • Edit Online

There are five settings for kernel-mode crash dump files:

Complete Memory Dump

Kernel Memory Dump

Small Memory Dump

Automatic Memory Dump

Active Memory Dump

The difference between these dump files is one of size. The *Complete Memory Dump* is the largest and contains the most information, including some User-Mode memory. The *Active Memory Dump* is somewhat smaller but contains similar information for most purposes. The *Kernel Memory Dump* is smaller still and typically omits User-Mode memory, and the *Small Memory Dump* is only 64 KB in size.

If you select *Automatic Memory Dump*, the dump file is the same as a Kernel Memory Dump, but Windows has more flexibility in setting the size of the system paging file.

The advantage to the larger files is that, since they contain more information, they are more likely to help you find the cause of the crash.

The advantage of the smaller files is that they are smaller and written more quickly. Speed is often valuable; if you are running a server, you may want the server to reboot as quickly as possible after a crash, and the reboot will not take place until the dump file has been written.

After a Complete Memory Dump or Kernel Memory Dump has been created, it is possible to create a Small Memory Dump file from the larger dump file. See the **.dump (Create Dump File)** command for details.

Note Much information can be obtained by analyzing a kernel-mode dump file. However, no kernel-mode dump file can provide as much information as actually debugging the crash directly with a kernel debugger.

Related topics

Kernel-Mode Dump Files

Enabling a Kernel-Mode Dump File

Complete Memory Dump

10/23/2019 • 2 minutes to read • Edit Online

A *Complete Memory Dump* is the largest kernel-mode dump file. This file 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.

Starting with Windows 8, you can register a *BugCheckAddPagesCallback* routine that is called during a complete memory dump. Your *BugCheckAddPagesCallback* routine can specify driver-specific data to add to the dump file. For example, 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.

This dump file requires a pagefile on your boot drive that is at least as large as your main system memory; it should be able to hold a file whose size equals your entire RAM plus one megabyte.

The Complete Memory Dump file is written to %SystemRoot%\Memory.dmp by default.

If a second bug check occurs and another Complete Memory Dump (or Kernel Memory Dump) is created, the previous file will be overwritten.

Related topics

Varieties of Kernel-Mode Dump Files

Kernel Memory Dump

12/5/2018 • 2 minutes to read • Edit Online

A *Kernel Memory Dump* contains all the memory in use by the kernel at the time of the crash.

This kind of dump file is significantly smaller than the Complete Memory Dump. Typically, the dump file will be around one-third the size of the physical memory on the system. This quantity will vary considerably, depending on your circumstances.

This dump file will not include unallocated memory, or any memory allocated to user-mode applications. It only includes memory allocated to the Windows kernel and hardware abstraction level (HAL), as well as memory allocated to kernel-mode drivers and other kernel-mode programs.

For most purposes, this crash dump is the most useful. It is significantly smaller than the Complete Memory Dump, but it only omits those portions of memory that are unlikely to have been involved in the crash.

Since this kind of dump file does not contain images of any user-mode executables residing in memory at the time of the crash, you may also need to set the executable image path if these executables turn out to be important.

The Kernel Memory Dump file is written to %SystemRoot%\Memory.dmp by default.

If a second bug check occurs and another Kernel Memory Dump (or Complete Memory Dump) is created, the previous file will be overwritten.

To suppress missing page error messages when debugging a Kernel Memory Dump, use the **.ignore_missing_pages** command.

Related topics

Varieties of Kernel-Mode Dump Files

Small Memory Dump

12/5/2018 • 2 minutes to read • Edit Online

A *Small Memory Dump* is much smaller than the other two kinds of kernel-mode crash dump files. It is exactly 64 KB in size, and requires only 64 KB of pagefile space on the boot drive.

This dump file includes the following:

- The bug check message and parameters, as well as other blue-screen data.
- The processor context (PRCB) for the processor that crashed.
- The process information and kernel context (EPROCESS) for the process that crashed.
- The thread information and kernel context (ETHREAD) for the thread that crashed.
- The kernel-mode call stack for the thread that crashed. If this is longer than 16 KB, only the topmost 16 KB will be included.
- A list of loaded drivers.

In Windows XP and later versions of Windows, the following items are also included:

- A list of loaded modules and unloaded modules.
- The debugger data block. This contains basic debugging information about the system.
- Any additional memory pages that Windows identifies as being useful in debugging failures. This includes the data pages that the registers were pointing to when the crash occurred, and other pages specifically requested by the faulting component.
- (Windows Server 2003 and later) The Windows SKU -- for example, "Professional" or "Server".

This kind of dump file can be useful when space is greatly limited. However, due to the limited amount of information included, errors that were not directly caused by the thread executing at time of crash may not be discovered by an analysis of this file.

Since this kind of dump file does not contain images of any executables residing in memory at the time of the crash, you may also need to set the executable image path if these executables turn out to be important.

If a second bug check occurs and a second Small Memory Dump file is created, the previous file will be preserved. Each additional file will be given a distinct name, which contains the date of the crash encoded in the filename. For example, mini022900-01.dmp is the first memory dump file generated on February 29, 2000. A list of all Small Memory Dump files is kept in the directory %SystemRoot%\Minidump.

Related topics

Varieties of Kernel-Mode Dump Files

Automatic Memory Dump

12/5/2018 • 2 minutes to read • Edit Online

An *Automatic Memory Dump* contains the same information as a Kernel Memory Dump. The difference between the two is not in the dump file itself, but in the way that Windows sets the size of the system paging file.

If the system paging file size is set to **System managed size**, and the kernel-mode crash dump is set to **Automatic Memory Dump**, then Windows can set the size of the paging file to less than the size of RAM. In this case, Windows sets the size of the paging file large enough to ensure that a kernel memory dump can be captured most of the time.

If the computer crashes and the paging file is not large enough to capture a kernel memory dump, Windows increases the size of the paging file to at least the size of RAM. The time of this event is recorded here in the Registry:

HKLM\SYSTEM\CurrentControlSet\Control\CrashControl\LastCrashTime

The increased paging file size stays in place for 4 weeks and then returns to the smaller size. If you want to return to the smaller paging file before 4 weeks, you can delete the Registry entry.

To see the paging file settings, go to **Control Panel > System and Security > System > Advanced system settings**. Under **Performance**, click **Settings**. On the **Advanced** tab, under **Virtual memory**, click **Change**. In the Virtual Memory dialog box, you can see the paging file settings.



The Automatic Memory Dump file is written to %SystemRoot%\Memory.dmp by default.

The Automatic Memory Dump is available in Windows 8 and later.

Note To suppress missing page error messages when debugging an Automatic Memory Dump, use the **.ignore_missing_pages** command.

Related topics

Varieties of Kernel-Mode Dump Files

Kernel-Mode Dump Files

Creating a Kernel-Mode Dump File

Active Memory Dump

12/5/2018 • 2 minutes to read • Edit Online

An *Active Memory Dump* is similar to a Complete Memory Dump, but it filters out pages that are not likely to be relevant to troubleshooting problems on the host machine. Because of this filtering, it is typically significantly smaller than a complete memory dump.

This dump file does include any memory allocated to user-mode applications. It also includes memory allocated to the Windows kernel and hardware abstraction level (HAL), as well as memory allocated to kernel-mode drivers and other kernel-mode programs. The dump includes active pages mapped into the kernel or user space that are useful for debugging, as well as selected Pagefile-backed Transition, Standby, and Modified pages such as the memory allocated with VirtualAlloc or page-file backed sections. Active dumps do not include pages on the free and zeroed lists, the file cache, guest VM pages and various other types of memory that are not likely to be useful during debugging.

An Active Memory Dump is particularly useful when Windows is hosting virtual machines (VMs). When taking a complete memory dump, the contents of each VM is included. When there are multiple VMs running, this can account for a large amount of memory in use on the host system. Many times, the code activities of interest are in the parent host OS, not the child VMs. An active memory dump filters out the memory associated with all of the child VMs.

The Active Memory Dump file is written to %SystemRoot%\Memory.dmp by default.

The Active Memory Dump is available in Windows 10 and later.

Note To suppress missing page error messages when debugging an Active Memory Dump, use the **.ignore_missing_pages** command.

Related topics

Varieties of Kernel-Mode Dump Files

Creating a Kernel-Mode Dump File

12/5/2018 • 2 minutes to read • Edit Online

There are three ways in which a kernel-mode dump file can be created:

1. You can enable the dump file from the Control Panel, and then the system can crash on its own.
2. You can enable the dump file from the Control Panel, and then force the system to crash.
3. You can use the debugger to create a dump file without crashing the system.

This section includes:

Enabling a Kernel-Mode Dump File

Forcing a System Crash

Creating a Dump File Without a System Crash

Verifying the Creation of a Kernel-Mode Dump File

Using an NMI Switch

It is also possible to use an NMI switch to create a crash dump file. Contact your hardware vendor to determine whether your machine has this switch.

The usage of NMI switches is not covered in this documentation.

Enabling a Kernel-Mode Dump File

12/5/2018 • 2 minutes to read • Edit Online

During a system crash, the Windows crash dump settings determine whether a dump file will be created, and if so, what size the dump file will be.

The Windows Control Panel controls the kernel-mode crash dump settings. Only a system administrator can modify these settings.

To change these settings, go to **Control Panel > System and Security > System**. Click **Advanced system settings**. Under **Startup and Recovery**, click **Settings**.

You will see the following dialog box:



Under **Write Debugging Information**, you can specify a kernel-mode dump file setting. Only one dump file can be created for any given crash. See Varieties of Kernel-Mode Dump Files for a description of different dump file settings.

You can also select or deselect the **Write an event to the system log** and **Automatically restart** options.

The settings that you select will apply to any kernel-mode dump file created by a system crash, regardless of whether the system crash was accidental or whether it was caused by the debugger. See Forcing a System Crash for details on causing a deliberate crash.

However, these settings do not affect dump files created by the **.dump** command. See Creating a Dump File Without a System Crash for details on using this command.

Related topics

Kernel-Mode Dump Files

Varieties of Kernel-Mode Dump Files

Forcing a System Crash

12/5/2018 • 2 minutes to read • Edit Online

Once kernel-mode dump files have been enabled, most system crashes should cause a crash file to be written and the blue screen to be displayed.

However, there are times that a system freezes without actually initiating a kernel crash. Possible symptoms of such a freeze include:

- The mouse pointer moves, but can't do anything.
- All video is frozen, the mouse pointer does not move, but paging continues.
- There is no response at all to the mouse or keyboard, and no use of the disk.

If an experienced debugging technician is present, he or she can hook up a kernel debugger and analyze the problem. For some tips on what to look for when this situation occurs, see Debugging a Stalled System.

However, if no technician is present, you may wish to create a kernel-mode dump file and send it to an off-site technician. This dump file can be used to analyze the cause of the error.

There are two ways to deliberately cause a system crash:

Forcing a System Crash from the Debugger

Forcing a System Crash from the Keyboard

Forcing a System Crash from the Debugger

12/5/2018 • 2 minutes to read • Edit Online

If KD or WinDbg is performing kernel-mode debugging, it can force a system crash to occur. This is done by entering the **.crash (Force System Crash)** command at the command prompt. (If the target computer does not crash immediately, follow this with the **g (Go)** command.)

When this command is issued, the system will call **KeBugCheck** and issue **bug check 0xE2** (MANUALLY_INITIATED_CRASH). Unless crash dumps have been disabled, a crash dump file is written at this point.

After the crash dump file has been written, the kernel debugger on the host computer will be alerted and can be used to actively debug the crashed target.

Forcing a System Crash from the Keyboard

7/1/2019 • 3 minutes to read • Edit Online

The following types of keyboards can cause a system crash directly:

PS/2 keyboards connected on i8042prt ports

This feature is available in Windows 2000 and later versions of Windows operating system.

USB keyboards

This feature is available in Windows Vista and later versions of Windows operating system.

Hyper-V keyboards

This feature is available in Windows 10 version 1903 and later versions of Windows operating system.

Configuration

Configure the following settings to enable the a system crash using keyboard:

1. If you wish a crash dump file to be written, you must enable such dump files, choose the path and file name, and select the size of the dump file. For more information, see Enabling a Kernel-Mode Dump File.
2. With PS/2 keyboards, you must enable the keyboard-initiated crash in the registry. In the registry key **HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\i8042prt\Parameters**, create a value named **CrashOnCtrlScroll**, and set it equal to a REG_DWORD value of 0x01.
3. With USB keyboards, you must enable the keyboard-initiated crash in the registry. In the registry key **HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\kbdhid\Parameters,** create a value named **CrashOnCtrlScroll**, and set it equal to a REG_DWORD value of 0x01.
4. With Hyper-V keyboards, you must enable the keyboard-initiated crash in the registry. In the registry key **HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\hyperkbd\Parameters**, create a value named **CrashOnCtrlScroll**, and set it equal to a REG_DWORD value of 0x01.

You must restart the system for these settings to take effect.

After this is completed, the keyboard crash can be initiated by using the following hotkey sequence: Hold down the rightmost CTRL key, and press the SCROLL LOCK key twice.

The system then calls **KeBugCheck** and issues **bug check 0xE2** (MANUALLY_INITIATED_CRASH). Unless crash dumps have been disabled, a crash dump file is written at this point.

If a kernel debugger is attached to the crashed machine, the machine will break into the kernel debugger after the crash dump file has been written.

For more information on using this feature, refer to the article Generate a memory dump file by using the keyboard (KB 244139).

Defining Alternate Keyboard Shortcuts to Force a System Crash from the Keyboard

You can configure values under the following registry subkeys for different keyboard shortcut sequences to generate the memory dump file:

- For PS/2 keyboards:

  HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\i8042prt\crashdump

- For USB keyboards:

  HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\kbdhid\crashdump

- For Hyper-V keyboards:

  HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\hyperkbd\crashdump

You must create the following registry REG_DWORD values under these subkeys:

Dump1Keys
The **Dump1Keys** registry value is a bit map of the first hot key to use. For example, instead of using the rightmost CTRL key to initiate the hot key sequence, you can set the first hot key to be the leftmost SHIFT key.

The values for the first hot key are described in the following table.

VALUE	FIRST KEY USED IN THE KEYBOARD SHORTCUT SEQUENCE
0x01	Rightmost SHIFT key
0x02	Rightmost CTRL key
0x04	Rightmost ALT key
0x10	Leftmost SHIFT key
0x20	Leftmost CTRL key
0x40	Leftmost ALT key

Note You can assign **Dump1Keys** a value that enables one or more keys as the first key used in the keyboard shortcut sequence. For example, assign **Dump1Keys** a value of 0x11 to define both the rightmost and leftmost SHIFT keys as the first key in the keyboard shortcut sequence.

Dump2Key
The **Dump2Key** registry value is the index into the scancode table for the keyboard layout of the target computer. The following is the actual table in the driver.

```
const UCHAR keyToScanTbl[134] = {
        0x00,0x29,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09,
        0x0A,0x0B,0x0C,0x0D,0x7D,0x0E,0x0F,0x10,0x11,0x12,
        0x13,0x14,0x15,0x16,0x17,0x18,0x19,0x1A,0x1B,0x00,
        0x3A,0x1E,0x1F,0x20,0x21,0x22,0x23,0x24,0x25,0x26,
        0x27,0x28,0x2B,0x1C,0x2A,0x00,0x2C,0x2D,0x2E,0x2F,
        0x30,0x31,0x32,0x33,0x34,0x35,0x73,0x36,0x1D,0x00,
        0x38,0x39,0xB8,0x00,0x9D,0x00,0x00,0x00,0x00,0x00,
        0x00,0x00,0x00,0x00,0x00,0xD2,0xD3,0x00,0x00,0xCB,
        0xC7,0xCF,0x00,0xC8,0xD0,0xC9,0xD1,0x00,0x00,0xCD,
        0x45,0x47,0x4B,0x4F,0x00,0xB5,0x48,0x4C,0x50,0x52,
        0x37,0x49,0x4D,0x51,0x53,0x4A,0x4E,0x00,0x9C,0x00,
        0x01,0x00,0x3B,0x3C,0x3D,0x3E,0x3F,0x40,0x41,0x42,
        0x43,0x44,0x57,0x58,0x00,0x46,0x00,0x00,0x00,0x00,
        0x00,0x7B,0x79,0x70 };
```

Note Index 124 (sysreq) is a special case because an 84-key keyboard has a different scan code.

If you define alternate keyboard shortcuts to force a system crash from a USB or PS/2 keyboard, you must either set the **CrashOnCtrlScroll** registry value to 0 or remove it from the registry.

Limitations

It is possible for a system to freeze in such a way that the keyboard shortcut sequence will not work. However, this should be a very rare occurrence. Using the keyboard shortcut sequence to initiate a crash will work even in many instances where CTRL+ALT+DELETE does not work.

Forcing a system crash from the keyboard does not work if the computer stops responding at a high interrupt request level (IRQL). This limitation exists because the Kbdhid.sys driver, which allows the memory dump process to run, operates at a lower IRQL than the i8042prt.sys driver.

Creating a Dump File Without a System Crash

12/5/2018 • 2 minutes to read • Edit Online

If KD or WinDbg is performing kernel-mode debugging, it can cause a kernel-mode dump file to be written without crashing the target computer.

This dump file can be either a Complete Memory Dump or a Small Memory Dump. The Control Panel settings are not relevant to this action.

Whereas dump files caused by a system crash are written to the computer that has crashed, this dump file will be written to the host computer.

For details, see the **.dump (Create Dump File)** command.

Verifying the Creation of a Kernel-Mode Dump File

12/5/2018 • 2 minutes to read • Edit Online

If you have a machine that has broken into the debugger, but you are unsure whether the crash dump file was successfully written, execute the following command:

```
dd nt!IopFinalCrashDumpStatus L1
```

This displays the value of the **IopFinalCrashDumpStatus** variable.

If this value equals zero, the process was successful. If it equals -1 (0xFFFFFFFF), the dump process has not started.

Any other value is a status code indicating an error during the dump process.

Analyzing a Kernel-Mode Dump File

12/5/2018 • 2 minutes to read • Edit Online

This section includes:

Analyzing a Kernel-Mode Dump File with KD

Analyzing a Kernel-Mode Dump File with WinDbg

Analyzing a Kernel-Mode Dump File with KAnalyze

Installing Symbol Files

Regardless of which tool you use, you need to install the symbol files for the version of Windows that generated the dump file. These files will be used by the debugger you choose to use to analyze the dump file. For more information about the proper installation of symbol files, see Installing Windows Symbol Files.

DumpExam

The DumpExam tool is obsolete. It is no longer needed in the analysis of a crash dump file.

Analyzing a Kernel-Mode Dump File with KD

8/27/2019 • 2 minutes to read • Edit Online

Kernel-mode memory dump files can be analyzed by KD. The processor or Windows version that the dump file was created on does not need to match the platform on which KD is being run.

Starting KD

To analyze a dump file, start KD with the **-z** command-line option:

kd -y *SymbolPath* **-i** *ImagePath* **-z** *DumpFileName*

The **-v** option (verbose mode) is also useful. For a full list of options, see **KD Command-Line Options**.

You can also open a dump file after the debugger is running by using the **.opendump (Open Dump File)** command, followed with **g (Go)**.

It is possible to debug multiple dump files at the same time. This can be done by including multiple **-z** switches on the command line (each followed by a different file name), or by using **.opendump** to add additional dump files as debugger targets. For information about how to control a multiple-target session, see Debugging Multiple Targets.

Dump files generally end with the extension .dmp or .mdmp. You can use network shares or Universal Naming Convention (UNC) file names for the memory dump file.

It is also common for dump files to be packed into a CAB file. If you specify the file name (including the .cab extension) after the **-z** option or as the argument to an **.opendump** command, the debugger can read the dump files directly out of the CAB. However, if there are multiple dump files stored in a single CAB, the debugger will only be able to read one of them. The debugger will not read any additional files from the CAB, even if they were symbol files or other files associated with the dump file.

Analyzing the Dump File

If you are analyzing a Kernel Memory Dump or a Small Memory Dump, you may need to set the executable image path to point to any executable files which may have been loaded in memory at the time of the crash.

Analysis of a dump file is similar to analysis of a live debugging session. See the Debugger Commands reference section for details on which commands are available for debugging dump files in kernel mode.

In most cases, you should begin by using **!analyze**. This extension command performs automatic analysis of the dump file and can often result in a lot of useful information.

The **.bugcheck (Display Bug Check Data)** shows the bug check code and its parameters. Look up this bug check in the Bug Check Code Reference for information about the specific error.

The following debugger extensions are especially useful for analyzing a kernel-mode crash dump:

lm

!kdext*.locks

!memusage

!vm

!errlog

!process 0 0

!process 0 7

For techniques that can be used to read specific kinds of information from a dump file, see Extracting Information from a Dump File.

Analyzing a Kernel-Mode Dump File with WinDbg

8/27/2019 • 2 minutes to read • Edit Online

Kernel-mode memory dump files can be analyzed by WinDbg. The processor or Windows version that the dump file was created on does not need to match the platform on which KD is being run.

Starting WinDbg

To analyze a dump file, start WinDbg with the **-z** command-line option:

windbg -y *SymbolPath* **-i** *ImagePath* **-z** *DumpFileName*

The **-v** option (verbose mode) is also useful. For a full list of options, see **WinDbg Command-Line Options**.

If WinDbg is already running and is in dormant mode, you can open a crash dump by selecting the **File | Open Crash Dump** menu command or pressing the CTRL+D shortcut key. When the **Open Crash Dump** dialog box appears, enter the full path and name of the crash dump file in the **File name** text box, or use the dialog box to select the proper path and file name. When the proper file has been chosen, click **Open**.

You can also open a dump file after the debugger is running by using the **.opendump (Open Dump File)** command, followed with **g (Go)**.

It is possible to debug multiple dump files at the same time. This can be done by including multiple **-z** switches on the command line (each followed by a different file name), or by using **.opendump** to add additional dump files as debugger targets. For information about how to control a multiple-target session, see Debugging Multiple Targets.

Dump files generally end with the extension .dmp or .mdmp. You can use network shares or Universal Naming Convention (UNC) file names for the memory dump file.

It is also common for dump files to be packed into a CAB file. If you specify the file name (including the .cab extension) after the **-z** option or as the argument to an **.opendump** command, the debugger can read the dump files directly out of the CAB. However, if there are multiple dump files stored in a single CAB, the debugger will only be able to read one of them. The debugger will not read any additional files from the CAB, even if they were symbol files or other files associated with the dump file.

Analyzing the Dump File

If you are analyzing a Kernel Memory Dump or a Small Memory Dump, you may need to set the executable image path to point to any executable files that may have been loaded in memory at the time of the crash.

Analysis of a dump file is similar to analysis of a live debugging session. See the Debugger Commands reference section for details on which commands are available for debugging dump files in kernel mode.

In most cases, you should begin by using **!analyze**. This extension command performs automatic analysis of the dump file and can often result in a lot of useful information.

The **.bugcheck (Display Bug Check Data)** shows the bug check code and its parameters. Look up this bug check in the Bug Check Code Reference for information about the specific error.

The following debugger extensions are especially useful for analyzing a kernel-mode crash dump:

lm

!kdext*.locks

!memusage

!vm

!errlog

!process 0 0

!process 0 7

For techniques that can be used to read specific kinds of information from a dump file, see Extracting Information from a Dump File.

Analyzing a Kernel-Mode Dump File with KAnalyze

12/5/2018 • 2 minutes to read • Edit Online

Kernel Memory Space Analyzer (KAnalyze, Kanalyze.exe) is another tool that can examine kernel-mode dump files.

KAnalyze and its documentation are part of the OEM Support Tools package.

To download these tools, go to Microsoft Support Article 253066 and follow the instructions on that page.

User-Mode Dump Files

10/8/2019 • 6 minutes to read • Edit Online

This section includes:

- Varieties of User-Mode Dump Files
- Creating a User-Mode Dump File

For information on analyzing a dump file, see Analyzing a User-Mode Dump File.

Varieties of User-Mode Dump Files

There are several kinds of user-mode crash dump files, but they are divided into two categories:

Full User-Mode Dumps

Minidumps

The difference between these dump files is one of size. Minidumps are usually more compact, and can be easily sent to an analyst.

Note Much information can be obtained by analyzing a dump file. However, no dump file can provide as much information as actually debugging the crash directly with a debugger.

Full User-Mode Dumps

A *full user-mode dump* is the basic user-mode dump file.

This dump file includes the entire memory space of a process, the program's executable image itself, the handle table, and other information that will be useful to the debugger in reconstructing the memory that was in use when the dump occurred.

It is possible to "shrink" a full user-mode dump file into a minidump. Simply load the dump file into the debugger and then use the **.dump (Create Dump File)** command to save a new dump file in minidump format.

Note Despite their names, the largest "minidump" file actually contains more information than the full user-mode dump. For example, **.dump /mf** or **.dump /ma** will create a larger and more complete file than **.dump /f**.

In user mode, **.dump /m[***MiniOptions***]** is the best choice. The dump files created with this switch can vary in size from very small to very large. By specifying the proper *MiniOptions* you can control exactly what information is included.

Minidumps

A user-mode dump file that includes only selected parts of the memory associated with a process is called a *minidump*.

The size and contents of a minidump file vary depending on the program being dumped and the application doing the dumping. Sometimes, a minidump file is fairly large and includes the full memory and handle table. Other times, it is much smaller -- for example, it might only contain information about a single thread, or only contain information about modules that are actually referenced in the stack.

The name "minidump" is misleading, because the largest minidump files actually contain more information than the "full" user-mode dump. For example, **.dump /mf** or **.dump /ma** will create a larger and more complete file

than **.dump /f**. For this reason, **.dump /m**[*MiniOptions*] recommended over **.dump /f** for all user-mode dump file creation.

If you are creating a minidump file with the debugger, you can choose exactly what information to include. A simple **.dump /m** command will include basic information about the loaded modules that make up the target process, thread information, and stack information. This can be modified by using any of the following options:

.DUMP OPTION	EFFECT ON DUMP FILE
/ma	Creates a minidump with all optional additions. The **/ma** option is equivalent to **/mfFhut** -- it adds full memory data, handle data, unloaded module information, basic memory information, and thread time information to the minidump.
/mf	Adds full memory data to the minidump. All accessible committed pages owned by the target application will be included.
/mF	Adds all basic memory information to the minidump. This adds a stream to the minidump that contains all basic memory information, not just information about valid memory. This allows the debugger to reconstruct the complete virtual memory layout of the process when the minidump is being debugged.
/mh	Adds data about the handles associated with the target application to the minidump.
/mu	Adds unloaded module information to the minidump. This is only available in Windows Server 2003 and later versions of Windows.
/mt	Adds additional thread information to the minidump. This includes thread times, which can be displayed by using **.ttime (Display Thread Times)** when debugging the minidump.
/mi	Adds *secondary memory* to the minidump. Secondary memory is any memory referenced by a pointer on the stack or backing store, plus a small region surrounding this address.
/mp	Adds process environment block (PEB) and thread environment block (TEB) data to the minidump. This can be useful if you need access to Windows system information regarding the application's processes and threads.
/mw	Adds all committed read-write private pages to the minidump.

.DUMP OPTION	EFFECT ON DUMP FILE
/md	Adds all read-write data segments within the executable image to the minidump.
/mc	Adds code sections within images.
/mr	Deletes from the minidump those portions of the stack and store memory that are not useful for recreating the stack trace. Local variables and other data type values are deleted as well. This option does not make the minidump smaller (since these memory sections are simply zeroed), but it is useful if you wish to protect the privacy of other applications.
/mR	Deletes the full module paths from the minidump. Only the module *names* will be included. This is a useful option if you wish to protect the privacy of the user's directory structure.
/mk " *FileName* **"**	(Windows Vista only) Creates a kernel-mode minidump in addition to the user-mode minidump. The kernel-mode minidump will be restricted to the same threads that are stored in the user-mode minidump. *FileName* must be enclosed in quotation marks.

These options can be combined. For example, the command **.dump /mfiu** can be used to create a fairly large minidump, or the command **.dump /mrR** can be used to create a minidump that preserves the user's privacy. For full syntax details, see **.dump (Create Dump File)**.

Creating a User-Mode Dump File

There are several different tools that can be used to create a user-mode dump file: CDB, WinDbg, Windows Error Reporting (WER), UserDump, and ADPlus.

For information about creating a user-mode dump file through ADPlus, see ADPlus.

For information about creating a user-mode dump file through WER, see Windows Error Reporting.

Choosing the Best Tool

There are several different tools that can create user-mode dump files. In most cases, ADPlus is the best tool to use.

The following table shows the features of each tool.

FEATURE	ADPLUS	WINDOWS ERROR REPORTING	CDB AND WINDBG	USERDUMP
Creating a dump file when an application crashes (postmortem debugging)	Yes	Yes	Yes	Yes
Creating a dump file when an application "hangs" (stops responding but does not actually crash)	Yes	No	Yes	Yes
Creating a dump file when an application encounters an exception	Yes	Yes	Yes	Yes
Creating a dump file while an application is running normally	No	No	Yes	No
Creating a dump file from an application that fails during startup	No	No	Yes	Yes
Shrinking an existing dump file	No	No	Yes	No

CDB and WinDbg

CDB and WinDbg can create user-mode dump files in a variety of ways.

Creating a Dump File Automatically

When an application error occurs, Windows can respond in several different ways, depending on the postmortem debugging settings. If these settings instruct a debugging tool to create a dump file, a user-mode memory dump file will be created. For more information, see Enabling Postmortem Debugging.

Creating Dump Files While Debugging

When CDB or WinDbg is debugging a user-mode application, you can also the **.dump (Create Dump File)** command to create a dump file.

This command does not cause the target application to terminate. By selecting the proper command options, you can create a minidump file that contains exactly the amount of information you wish.

Shrinking an Existing Dump File

CDB and WinDbg can also be used to *shrink* a dump file. To do this, begin debugging an existing dump file, and then use the **.dump** command to create a dump file of smaller size.

UserDump

The UserDump tool (Userdump.exe), also known as User-Mode Process Dump, can create user-mode dump files.

UserDump and its documentation are part of the OEM Support Tools package.

For more info and to download these tools, see How to use the Userdump.exe tool to create a dump file and follow the instructions on that page. Additionally, When CDB or WinDbg is debugging a user-mode application, you can also use the .dump (Create Dump File) command to create a dump file.

Analyzing a User-Mode Dump File

12/5/2018 • 5 minutes to read • Edit Online

This topic includes:

- Analyzing a User-Mode Dump File with WinDbg
- Analyzing a User-Mode Dump File with CDB

Analyzing a User-Mode Dump File with WinDbg

User-mode memory dump files can be analyzed by WinDbg. The processor or Windows version that the dump file was created on does not need to match the platform on which WinDbg is being run.

Installing Symbol Files

Before analyzing the memory dump file, you will need to install the symbol files for the version of Windows that generated the dump file. These files will be used by the debugger you choose to use to analyze the dump file. For more information about the proper installation of symbol files, see Installing Windows Symbol Files.

You will also need to install all the symbol files for the user-mode process, either an application or system service, that caused the system to generate the dump file. If this code was written by you, the symbol files should have been generated when the code was compiled and linked. If this is commercial code, check on the product CD-ROM or contact the software manufacturer for these particular symbol files.

Starting WinDbg

To analyze a dump file, start WinDbg with the **-z** command-line option:

windbg -y *SymbolPath* **-i** *ImagePath* **-z** *DumpFileName*

The **-v** option (verbose mode) is also useful. For a full list of options, see **WinDbg Command-Line Options**.

If WinDbg is already running and is in dormant mode, you can open a crash dump by selecting the **File | Open Crash Dump** menu command or pressing the CTRL+D shortcut key. When the **Open Crash Dump** dialog box appears, enter the full path and name of the crash dump file in the **File name** text box, or use the dialog box to select the proper path and file name. When the proper file has been chosen, click **Open**.

You can also open a dump file after the debugger is running by using the **.opendump (Open Dump File)** command, followed with **g (Go)**.

It is possible to debug multiple dump files at the same time. This can be done by including multiple **-z** switches on the command line (each followed by a different file name), or by using **.opendump** to add additional dump files as debugger targets. For information about how to control a multiple-target session, see Debugging Multiple Targets.

Dump files generally end with the extension .dmp or .mdmp. You can use network shares or Universal Naming Convention (UNC) file names for the memory dump file.

It is also common for dump files to be packed into a CAB file. If you specify the file name (including the .cab extension) after the **-z** option or as the argument to an **.opendump** command, the debugger can read the dump files directly out of the CAB. However, if there are multiple dump files stored in a single CAB, the debugger will only be able to read one of them. The debugger will not read any additional files from the CAB, even if they were symbol files or executables associated with the dump file.

Analyzing a Full User Dump File

Analysis of a full user dump file is similar to analysis of a live debugging session. See the Debugger Commands reference section for details on which commands are available for debugging dump files in user mode.

Analyzing Minidump Files

Analysis of a user-mode minidump file is done in the same way as a full user dump. However, since much less memory has been preserved, you are much more limited in the actions you can perform. Commands that attempt to access memory beyond what is preserved in the minidump file will not function properly.

Additional Techniques

For techniques that can be used to read specific kinds of information from a dump file, see Extracting Information from a Dump File.

Analyzing a User-Mode Dump File with CDB

User-mode memory dump files can be analyzed by CDB. The processor or Windows version that the dump file was created on does not need to match the platform on which CDB is being run.

Installing Symbol Files

Before analyzing the memory dump file, you will need to install the symbol files for the version of Windows that generated the dump file. These files will be used by the debugger you choose to use to analyze the dump file. For more information about the proper installation of symbol files, see Installing Windows Symbol Files.

You will also need to install all the symbol files for the user-mode process, either an application or system service, that caused the system to generate the dump file. If this code was written by you, the symbol files should have been generated when the code was compiled and linked. If this is commercial code, check on the product CD-ROM or contact the software manufacturer for these particular symbol files.

Starting CDB

To analyze a dump file, start CDB with the **-z** command-line option:

cdb -y *SymbolPath* **-i** *ImagePath* **-z** *DumpFileName*

The **-v** option (verbose mode) is also useful. For a full list of options, see **CDB Command-Line Options**.

You can also open a dump file after the debugger is running by using the **.opendump (Open Dump File)** command, followed with **g (Go)**. This allows you to debug multiple dump files at the same time.

It is possible to debug multiple dump files at the same time. This can be done by including multiple **-z** switches on the command line (each followed by a different file name), or by using **.opendump** to add additional dump files as debugger targets. For information about how to control a multiple-target session, see Debugging Multiple Targets.

Dump files generally end with the extension .dmp or .mdmp. You can use network shares or Universal Naming Convention (UNC) file names for the memory dump file.

It is also common for dump files to be packed into a CAB file. If you specify the file name (including the .cab extension) after the **-z** option or as the argument to an **.opendump** command, the debugger can read the dump files directly out of the CAB. However, if there are multiple dump files stored in a single CAB, the debugger will only be able to read one of them. The debugger will not read any additional files from the CAB, even if they are symbol files or executables associated with the dump file.

Analyzing a Full User Dump File

Analysis of a full user dump file is similar to analysis of a live debugging session. See the Debugger Commands reference section for details on which commands are available for debugging dump files in user mode.

Analyzing Minidump Files

Analysis of a user-mode minidump file is done in the same way as a full user dump. However, since much less

memory has been preserved, you are much more limited in the actions you can perform. Commands that attempt to access memory beyond what is preserved in the minidump file will not function properly.

Additional Techniques

For techniques that can be used to read specific kinds of information from a dump file, see Extracting Information from a Dump File.

Extracting Information from a Dump File

3/14/2019 • 2 minutes to read • Edit Online

Certain kinds of information, such as the name of the target computer, are easily available during live debugging. When debugging a dump file it takes a little more work to determine this information.

Finding the Computer Name in a Kernel-Mode Dump File

If you need to determine the name of the computer on which the crash dump was made, you can use the **!peb** extension and look for the value of COMPUTERNAME it its output.

Finding the IP Address in a Kernel-Mode Dump File

To determine the IP address of the computer on which the crash dump was made, find a thread stack that shows some send/receive network activity. Open one of the send packets or receive packets. The IP address will be visible in that packet.

Finding the Process ID in a User-Mode Dump File

To determine the process ID of the target application from a user-mode dump file, use the **| (Process Status)** command. This will display all the processes being debugged at the time the dump was written. The process marked with a period (**.**) is the current process. Its process ID is given in hexadecimal after the **id:** notation.

CAB Files that Contain Paging Files Along with a Memory Dump

12/5/2018 • 2 minutes to read • Edit Online

A memory dump file can be placed in a cabinet (CAB) file along with paging files. When a Windows debugger analyzes the memory dump file, it can use the paging files to present a full view memory, including memory that was paged out when the dump file was created.

Suppose a CAB file named MyCab.cab contains these files:

Memory.dmp Cabmanifest.xml Pagefile.sys Also suppose Cabmanifest.xml looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<WatsonPageFileManifest>
  <Pagefiles>
    <Pagefile Name="pagefile.sys"></Pagefile>
  </Pagefiles>
</WatsonPageFileManifest>
```

You can open the CAB file by entering one of these commands:

- **windbg /z MyCab.cab**
- **kd /z MyCab.cab**

The debugger reads Cabmanifest.xml for a list of paging files that are to be included in the debugging session. In this example, there is only one paging file. The debugger converts the paging file to a Target Information File (TIF) file that it can use during the debugging session. Because the debugger has access to the TIF, it can display memory that was paged out at the time the dump file was created.

Regardless of how many paging files are in the CAB file, the debugger uses only the paging files that are listed in Cabmanifest.xml. Here's an example of a CAB manifest file that lists three paging files.

```
<?xml version="1.0" encoding="UTF-8"?>
<WatsonPageFileManifest>
  <Pagefiles>
    <Pagefile Name="pagefile1.sys"></Pagefile>
    <Pagefile Name="pagefile2.sys"></Pagefile>
    <Pagefile Name="swapfile.sys"></Pagefile>
  </Pagefiles>
</WatsonPageFileManifest>
```

In Cabmanifest.xml, the paging files must be listed in the same order that Windows uses them. That is, they must be listed in the order that they appear in the Registry.

The memory dump file that you put in the CAB file must be a complete memory dump. You can use Control Panel to configure Windows to create a complete memory dump when there is a crash. For example, in Windows 8 you can go to **Control Panel > System and Security > System > Advanced System Settings > Startup and Recovery**. As an alternative to using Control Panel, you can set the value of this registry entry to 1.

HKLM\SYSTEM\CurrentControlSet\Control\CrashControl\CrashDumpEnabled

Starting in Windows 8.1, you can configure Windows to preserve the contents of paging files when Windows restarts.

To specify that you want paging files to be saved when Windows restarts, set the value of this registry entry to 1.

HKLM\SYSTEM\CurrentControlSet\Control\CrashControl\SavePageFileContents

Debugging OCA minidump files

6/25/2019 • 3 minutes to read • Edit Online

Online Crash Analysis (OCA) is the reporting facility for Windows Error Reporting (WER) information. Your company can use OCA crash dumps to analyze customer problems.

Analyze dump files

Dump files are a snapshot of the state of the computer (or process) at the time of the crash.

To analyze this data, a developer must use a debugger that can read user minidump files. The debugger must also have access to both the images and symbols that match the contents of the dump file. Most developers are aware of the need to use matching symbols when debugging a live crash. However, when debugging a minidump, matching images must also be available for the debugger.

Matching images must be available because minidump files store very little information; they store only some of the volatile information at the time of the crash. They do not store the basic code streams that the computer loaded into memory. Instead, to save space, the minidump file stores only the name and time stamp of the images loaded on the crashing computer.

To examine the code that was running on the crashing computer, the debugger must be given access to the same binaries that the crashing computer was running. The debugger uses the name and time stamp stored in the minidump file to uniquely match and load the binaries when the developer wants to debug the crash.

After the images and symbols are loaded in the debugger, you can analyze the state of the system at the time of the crash, including data that was saved after the crash occurred. The minidump does not, however, reproduce the steps that led to the specific failure. Finding the root cause requires analyzing the driver's source code to determine what code path may have led to the failure. Experience has shown that a large percentage of failures can be understood and addressed by analyzing dump files and source code.

Use symbols to match executable code with source code

The best way to access matching images and symbols is to use the Microsoft symbol server. Symbols are data that enable the debugger to map the executable code back to the source code. When you build a program, the program's symbols are usually stored in symbol files. When a debugger analyzes a program, it needs to access the program's symbols.

Symbol files can include any or all of the following:

- The names and addresses of all functions.
- All data type, structure, and class definitions.
- The names, data types, and addresses of global variables.
- The names, data types, addresses, and scopes of local variables.
- The line number in the source code that corresponds to each binary instruction.

The Windows Driver Kit (WDK) includes tools that can be used to reduce the number of symbols in a symbol file. The symbol files that contain all of the source-level information are called full symbol files. The symbol files with reduced information are called stripped symbol files.

Because symbol data is crucial for getting meaningful crash information from Windows Error Report (WER) data, we encourage you to submit your symbols when you submit drivers to be signed. When symbols are submitted, they are stored on a server that synchronizes symbol data with the associated WER processes. With this storage

process, you can easily categorize the crashes reported in the minidump files and ultimately receive better data back from Microsoft.

Microsoft provides a symbol server on the Internet that you can use to analyze the Windows modules that are present in minidump files. The server includes stripped symbol files for Windows and a few other products. Microsoft has added the binaries for Windows XP and Windows Server 2003. You can use the Internet symbol server and the Debugging Tools for Windows to analyze minidump files.

Integrate WER into applications

For more information on integrating WER into applications, see Using WER.

Related topics

Advanced Driver Debugging [336 KB] [PPT]

WDK and WinDbg downloads

Driver Debugging Basics [WinHEC 2007; 633 KB] [PPT]

How to read the small memory dump file that is created by Windows if a crash occurs

Resource-Definition Statements

Windows Error Reporting

VERSIONINFO resource

Bug Checks (Blue Screens)

12/5/2018 • 2 minutes to read • Edit Online

This section includes:

General Tips for Blue Screens

Blue Screen Data

Bug Check Code Reference

Note These topic are for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

General Tips for Blue Screens

12/5/2018 • 2 minutes to read • Edit Online

If your computer stops working and displays a blue screen, the computer has shut down abruptly to protect itself from data loss. A hardware device, its driver, or related software might have caused this error. If your copy of Windows came with your computer, you may want to contact the manufacturer of your computer.

If you have experience with computers and want to try to recover from this error, follow the steps described in Troubleshoot blue screen errors.

Blue Screen Data

6/25/2019 • 8 minutes to read • Edit Online

Note This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Note If you are an IT professional or support agent, see this article for additional information, Troubleshoot "blue screen" or Stop error problems before you contact Microsoft Support.

When Microsoft Windows encounters a condition that compromises safe system operation, the system halts. This condition is called a *bug check*. It is also commonly referred to as a *system crash*, a *kernel error*, or a *stop error*.

If the OS were allowed to continue to run after the operating system integrity is compromised, it could corrupt data or compromise the security of the system.

If crash dumps are enabled on the system, a crash dump file is created.

If a kernel debugger is attached and active, the system causes a break so that the debugger can be used to investigate the crash.

If no debugger is attached, a blue text screen appears with information about the error. This screen is called a *blue screen*, a *bug check screen*, or a *stop screen*.

If you are using an insider build of Windows, the text will be displayed on a green background.

The exact appearance of the blue screen depends on the cause of the error.

The following is an example of one possible blue screen:



The stop code is displayed such as **PAGE_FAULT_IN_NONPAGED_AREA**. When it is available, the module name of the code that was being executed is also displayed, such as AcmeVideo.sys.

If a kernel-mode dump file has been written, this will be indicated as well with a percentage complete count down as the dump is being written.

There is a stop code hex value associated with each stop code as listed in Bug Check Code Reference.

Gathering the Stop Code Parameters

Each bug check code has four associated parameters that provide additional information. The parameters are described in Bug Check Code Reference for each stop code.

There are multiple ways to gather the four stop code parameters.

- Examine the Windows system log in the event viewer. The event properties for the BugCheck will list the four stop code parameters. For more information, see Open Event Viewer.
- Load the generated dump file and use the **!analyze** command with the debugger attached. For more information, see Analyzing a Kernel-Mode Dump File with WinDbg.
- Attach a kernel debugger to the faulting PC. When the stop code occurs, the debugger output will include the four parameters after the stop code hex value.

```
*******************************************************************************
*                                                                             *
*                        Bugcheck Analysis                                    *
*                                                                             *
*******************************************************************************

Use !analyze -v to get detailed debugging information.

BugCheck 9F, {3, ffffe000f38c06a0, fffff803c596cad0, ffffe000f46a1010}

Implicit thread is now ffffe000`f4ca3040
Probably caused by : hidusb.sys
```

Bug Check Symbolic Names

DRIVER_POWER_STATE_FAILURE is the Bug Check Symbolic Name, with an associated bug check code of 9F. The stop code hex value associated with the Bug Check Symbolic Name is listed in the Bug Check Code Reference.

Reading Bug Check Information from the Debugger

If a debugger is attached, a bug check will cause the target computer to break into the debugger. In this case, the blue screen may not appear immediately, the full details on this crash will be sent to the debugger and appear in the debugger window. To see this information a second time, use the **.bugcheck (Display Bug Check Data)** command or the **!analyze** extension command.

Kernel Debugging and Crash Dump Analysis

Kernel debugging is especially useful when other troubleshooting techniques fail, or for a recurring problem. Remember to capture the exact text in the bug check information section of the error message. To isolate a complex problem and develop a viable workaround, it is useful to record the exact actions that lead to the failure.

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause.

You can also set a breakpoint in the code leading up to this stop code and attempt to single step forward into the faulting code.

For more information see the following topics:

Crash dump analysis using the Windows debuggers (WinDbg)

Analyzing a Kernel-Mode Dump File with WinDbg

Using the !analyze Extension and !analyze

The Defrag Tools show on Channel 9 - https://channel9.msdn.com/Shows/Defrag-Tools

Using Driver Verifier to Gather Information

It is estimated that about three quarters of blue screens are caused by faulting drivers. Driver Verifier is a tool that runs in real time to examine the behavior of drivers. For example, Driver Verifier checks the use of memory resources, such as memory pools. If it see errors in the execution of driver code, it proactively creates an exception to allow that part of the driver code to be further scrutinized. The driver verifier manager is built into Windows and is available on all Windows PCs. To start the driver verifier manager, type *Verifier* at a command prompt. You can configure which drivers you would like to verify. The code that verifies drivers adds overhead as it runs, so try and verify the smallest number of drivers as possible. For more information, see Driver Verifier.

Tips for Software Engineers

When a bug check occurs as a result of code you have written, you should use the kernel debugger to analyze the problem, and then fix the bugs in your code. For full details, see the individual bug check code in the Bug Check Code Reference section.

However, you might also encounter bug checks that are not caused by your own code. In this case, you probably will not be able to fix the actual cause of the problem, so your goal should be to work around the problem, and if possible isolate and remove the hardware or software component that is at fault.

Many problems can be resolved through basic troubleshooting procedures, such as verifying instructions, reinstalling key components, and verifying file dates. Also, the Event Viewer, the Sysinternals diagnostic tools and network monitoring tools might isolate and resolve these issues.

For general troubleshooting of Windows bug check codes, follow these suggestions:

- If you recently added hardware to the system, try removing or replacing it. Or check with the manufacturer to see if any patches are available.
- If new device drivers or system services have been added recently, try removing or updating them. Try to determine what changed in the system that caused the new bug check code to appear.
- Look in **Device Manager** to see if any devices are marked with the exclamation point (!). Review the events log displayed in driver properties for any faulting driver. Try updating the related driver.
- Check the System Log in Event Viewer for additional error messages that might help pinpoint the device or driver that is causing the error. For more information, see Open Event Viewer. Look for critical errors in the system log that occurred in the same time window as the blue screen.
- You can try running the hardware diagnostics supplied by the system manufacturer.
- Run the Windows Memory Diagnostics tool, to test the memory. In the control panel search box, type Memory, and then click **Diagnose your computer's memory problems**. After the test is run, use Event viewer to view the results under the System log. Look for the *MemoryDiagnostics-Results* entry to view the results.
- Confirm that any new hardware that is installed is compatible with the installed version of Windows. For example, you can get information about required hardware at Windows 10 Specifications.
- Run a virus detection program. Viruses can infect all types of hard disks formatted for Windows, and resulting disk corruption can generate system bug check codes. Make sure the virus detection program checks the Master Boot Record for infections.
- Use the scan disk utility to confirm that there are no file system errors. Right click on the drive you want to scan and select **Properties**. Click on **Tools**. Click the **Check now** button.
- Use the System File Checker tool to repair missing or corrupted system files. The System File Checker is a utility in Windows that allows users to scan for corruptions in Windows system files and restore corrupted files. Use the following command to run the System File Checker tool (SFC.exe).

```
SFC /scannow
```

For more information, see Use the System File Checker tool to repair missing or corrupted system files.

- Confirm that there is sufficient free space on the hard drive. The operating system and some applications require sufficient free space to create swap files and for other functions. Based on the system configuration, the exact requirement varies, but it is normally a good idea to have 10% to 15% free space available.
- Verify that the system has the latest Service Pack installed. To detect which Service Pack, if any, is installed on your system, click **Start**, click **Run**, type **winver**, and then press ENTER. The **About Windows** dialog box displays the Windows version number and the version number of the Service Pack, if one has been installed.
- Check with the manufacturer to see if an updated system BIOS or firmware is available.
- Disable BIOS memory options such as caching or shadowing.
- For PCs, make sure that all expansion boards are is properly seated and all cables are completely connected.

Using Safe Mode

Consider using Safe Mode when removing or disabling components. Using Safe Mode loads only the minimum required drivers and system services during the Windows startup. To enter Safe Mode, use **Update and Security** in Settings. Select **Recovery**->**Advanced startup** to boot to maintenance mode. At the resulting menu, choose **Troubleshoot**-> **Advanced Options** -> **Startup Settings** -> **Restart**. After Windows restarts to the **Startup Settings** screen, select option, 4, 5 or 6 to boot to Safe Mode.

Safe Mode may be available by pressing a function key on boot, for example F8. Refer to information from the manufacturer for specific startup options.

Forced KeBugCheck

To deliberately cause a bug check from a kernel-mode driver, you need to pass the bug check's symbolic name to the **KeBugCheck** or **KeBugCheckEx** function. This should only be done in circumstances where no other option is available. For more details on these functions, see the Windows Driver Kit.

Bug Check Code Reference

9/16/2019 • 4 minutes to read • Edit Online

This section contains descriptions of common bug check codes, including the parameters displayed with the error code on the blue bug check screen. This section also describes how you can diagnose the fault which led to the bug check, as well as possible ways to deal with the error.

> **NOTE**
>
> This topic is for programmers. If you are a customer whose system has displayed a blue screen with a bug check code, see Troubleshoot blue screen errors.

If a specific bug check code does not appear in this reference, use the **!analyze** extension in the Windows Debugger (WinDbg) with the following syntax (in kernel mode), replacing `<code>` with a bug check code:

```
!analyze -show <code>
```

Entering this command causes WinDbg to display information about the specified bug check code. If your default number base (radix) is not 16, prefix `<code>` with **0x**.

The following table shows the code and name of each bug check code.

CODE	NAME
0x00000001	**APC_INDEX_MISMATCH**
0x00000002	**DEVICE_QUEUE_NOT_BUSY**
0x00000003	**INVALID_AFFINITY_SET**
0x00000004	**INVALID_DATA_ACCESS_TRAP**
0x00000005	**INVALID_PROCESS_ATTACH_ATTEMPT**
0x00000006	**INVALID_PROCESS_DETACH_ATTEMPT**
0x00000007	**INVALID_SOFTWARE_INTERRUPT**
0x00000008	**IRQL_NOT_DISPATCH_LEVEL**
0x00000009	**IRQL_NOT_GREATER_OR_EQUAL**
0x0000000A	**IRQL_NOT_LESS_OR_EQUAL**
0x0000000B	**NO_EXCEPTION_HANDLING_SUPPORT**
0x0000000C	**MAXIMUM_WAIT_OBJECTS_EXCEEDED**
0x0000000D	**MUTEX_LEVEL_NUMBER_VIOLATION**

CODE	NAME
0x0000000E	NO_USER_MODE_CONTEXT
0x0000000F	SPIN_LOCK_ALREADY_OWNED
0x00000010	SPIN_LOCK_NOT_OWNED
0x00000011	THREAD_NOT_MUTEX_OWNER
0x00000012	TRAP_CAUSE_UNKNOWN
0x00000013	EMPTY_THREAD_REAPER_LIST
0x00000014	CREATE_DELETE_LOCK_NOT_LOCKED
0x00000015	LAST_CHANCE_CALLED_FROM_KMODE
0x00000016	CID_HANDLE_CREATION
0x00000017	CID_HANDLE_DELETION
0x00000018	REFERENCE_BY_POINTER
0x00000019	BAD_POOL_HEADER
0x0000001A	MEMORY_MANAGEMENT
0x0000001B	PFN_SHARE_COUNT
0x0000001C	PFN_REFERENCE_COUNT
0x0000001D	NO_SPIN_LOCK_AVAILABLE
0x0000001E	KMODE_EXCEPTION_NOT_HANDLED
0x0000001F	SHARED_RESOURCE_CONV_ERROR
0x00000020	KERNEL_APC_PENDING_DURING_EXIT
0x00000021	QUOTA_UNDERFLOW
0x00000022	FILE_SYSTEM
0x00000023	FAT_FILE_SYSTEM
0x00000024	NTFS_FILE_SYSTEM
0x00000025	NPFS_FILE_SYSTEM
0x00000026	CDFS_FILE_SYSTEM

CODE	NAME
0x00000027	RDR_FILE_SYSTEM
0x00000028	CORRUPT_ACCESS_TOKEN
0x00000029	SECURITY_SYSTEM
0x0000002A	INCONSISTENT_IRP
0x0000002B	PANIC_STACK_SWITCH
0x0000002C	PORT_DRIVER_INTERNAL
0x0000002D	SCSI_DISK_DRIVER_INTERNAL
0x0000002E	DATA_BUS_ERROR
0x0000002F	INSTRUCTION_BUS_ERROR
0x00000030	SET_OF_INVALID_CONTEXT
0x00000031	PHASE0_INITIALIZATION_FAILED
0x00000032	PHASE1_INITIALIZATION_FAILED
0x00000033	UNEXPECTED_INITIALIZATION_CALL
0x00000034	CACHE_MANAGER
0x00000035	NO_MORE_IRP_STACK_LOCATIONS
0x00000036	DEVICE_REFERENCE_COUNT_NOT_ZERO
0x00000037	FLOPPY_INTERNAL_ERROR
0x00000038	SERIAL_DRIVER_INTERNAL
0x00000039	SYSTEM_EXIT_OWNED_MUTEX
0x0000003A	SYSTEM_UNWIND_PREVIOUS_USER
0x0000003B	SYSTEM_SERVICE_EXCEPTION
0x0000003C	INTERRUPT_UNWIND_ATTEMPTED
0x0000003D	INTERRUPT_EXCEPTION_NOT_HANDLED
0x0000003E	MULTIPROCESSOR_CONFIGURATION_NOT_SUPPORTED
0x0000003F	NO_MORE_SYSTEM_PTES

CODE	NAME
0x00000040	TARGET_MDL_TOO_SMALL
0x00000041	MUST_SUCCEED_POOL_EMPTY
0x00000042	ATDISK_DRIVER_INTERNAL
0x00000043	NO_SUCH_PARTITION
0x00000044	MULTIPLE_IRP_COMPLETE_REQUESTS
0x00000045	INSUFFICIENT_SYSTEM_MAP_REGS
0x00000046	DEREF_UNKNOWN_LOGON_SESSION
0x00000047	REF_UNKNOWN_LOGON_SESSION
0x00000048	CANCEL_STATE_IN_COMPLETED_IRP
0x00000049	PAGE_FAULT_WITH_INTERRUPTS_OFF
0x0000004A	IRQL_GT_ZERO_AT_SYSTEM_SERVICE
0x0000004B	STREAMS_INTERNAL_ERROR
0x0000004C	FATAL_UNHANDLED_HARD_ERROR
0x0000004D	NO_PAGES_AVAILABLE
0x0000004E	PFN_LIST_CORRUPT
0x0000004F	NDIS_INTERNAL_ERROR
0x00000050	PAGE_FAULT_IN_NONPAGED_AREA
0x00000051	REGISTRY_ERROR
0x00000052	MAILSLOT_FILE_SYSTEM
0x00000053	NO_BOOT_DEVICE
0x00000054	LM_SERVER_INTERNAL_ERROR
0x00000055	DATA_COHERENCY_EXCEPTION
0x00000056	INSTRUCTION_COHERENCY_EXCEPTION
0x00000057	XNS_INTERNAL_ERROR
0x00000058	FTDISK_INTERNAL_ERROR

CODE	NAME
0x00000059	**PINBALL_FILE_SYSTEM**
0x0000005A	**CRITICAL_SERVICE_FAILED**
0x0000005B	**SET_ENV_VAR_FAILED**
0x0000005C	**HAL_INITIALIZATION_FAILED**
0x0000005D	**UNSUPPORTED_PROCESSOR**
0x0000005E	**OBJECT_INITIALIZATION_FAILED**
0x0000005F	**SECURITY_INITIALIZATION_FAILED**
0x00000060	**PROCESS_INITIALIZATION_FAILED**
0x00000061	**HAL1_INITIALIZATION_FAILED**
0x00000062	**OBJECT1_INITIALIZATION_FAILED**
0x00000063	**SECURITY1_INITIALIZATION_FAILED**
0x00000064	**SYMBOLIC_INITIALIZATION_FAILED**
0x00000065	**MEMORY1_INITIALIZATION_FAILED**
0x00000066	**CACHE_INITIALIZATION_FAILED**
0x00000067	**CONFIG_INITIALIZATION_FAILED**
0x00000068	**FILE_INITIALIZATION_FAILED**
0x00000069	**IO1_INITIALIZATION_FAILED**
0x0000006A	**LPC_INITIALIZATION_FAILED**
0x0000006B	**PROCESS1_INITIALIZATION_FAILE**D
0x0000006C	**REFMON_INITIALIZATION_FAILED**
0x0000006D	**SESSION1_INITIALIZATION_FAILED**
0x0000006E	**SESSION2_INITIALIZATION_FAILED**
0x0000006F	**SESSION3_INITIALIZATION_FAILED**
0x00000070	**SESSION4_INITIALIZATION_FAILED**
0x00000071	**SESSION5_INITIALIZATION_FAILED**

CODE	NAME
0x00000072	ASSIGN_DRIVE_LETTERS_FAILED
0x00000073	CONFIG_LIST_FAILED
0x00000074	BAD_SYSTEM_CONFIG_INFO
0x00000075	CANNOT_WRITE_CONFIGURATION
0x00000076	PROCESS_HAS_LOCKED_PAGES
0x00000077	KERNEL_STACK_INPAGE_ERROR
0x00000078	PHASE0_EXCEPTION
0x00000079	MISMATCHED_HAL
0x0000007A	KERNEL_DATA_INPAGE_ERROR
0x0000007B	INACCESSIBLE_BOOT_DEVICE
0x0000007C	BUGCODE_NDIS_DRIVER
0x0000007D	INSTALL_MORE_MEMORY
0x0000007E	SYSTEM_THREAD_EXCEPTION_NOT_HANDLED
0x0000007F	UNEXPECTED_KERNEL_MODE_TRAP
0x00000080	NMI_HARDWARE_FAILURE
0x00000081	SPIN_LOCK_INIT_FAILURE
0x00000082	DFS_FILE_SYSTEM
0x00000085	SETUP_FAILURE
0x0000008B	MBR_CHECKSUM_MISMATCH
0x0000008E	KERNEL_MODE_EXCEPTION_NOT_HANDLED
0x0000008F	PP0_INITIALIZATION_FAILED
0x00000090	PP1_INITIALIZATION_FAILED
0x00000092	UP_DRIVER_ON_MP_SYSTEM
0x00000093	INVALID_KERNEL_HANDLE
0x00000094	KERNEL_STACK_LOCKED_AT_EXIT

CODE	NAME
0x00000096	INVALID_WORK_QUEUE_ITEM
0x00000097	BOUND_IMAGE_UNSUPPORTED
0x00000098	END_OF_NT_EVALUATION_PERIOD
0x00000099	INVALID_REGION_OR_SEGMENT
0x0000009A	SYSTEM_LICENSE_VIOLATION
0x0000009B	UDFS_FILE_SYSTEM
0x0000009C	MACHINE_CHECK_EXCEPTION
0x0000009E	USER_MODE_HEALTH_MONITOR
0x0000009F	DRIVER_POWER_STATE_FAILURE
0x000000A0	INTERNAL_POWER_ERROR
0x000000A1	PCI_BUS_DRIVER_INTERNAL
0x000000A2	MEMORY_IMAGE_CORRUPT
0x000000A3	ACPI_DRIVER_INTERNAL
0x000000A4	CNSS_FILE_SYSTEM_FILTER
0x000000A5	ACPI_BIOS_ERROR
0x000000A7	BAD_EXHANDLE
0x000000AB	SESSION_HAS_VALID_POOL_ON_EXIT
0x000000AC	HAL_MEMORY_ALLOCATION
0x000000AD	VIDEO_DRIVER_DEBUG_REPORT_REQUEST
0x000000B1	BGI_DETECTED_VIOLATION
0x000000B4	VIDEO_DRIVER_INIT_FAILURE
0x000000B8	ATTEMPTED_SWITCH_FROM_DPC
0x000000B9	CHIPSET_DETECTED_ERROR
0x000000BA	SESSION_HAS_VALID_VIEWS_ON_EXIT
0x000000BB	NETWORK_BOOT_INITIALIZATION_FAILED

CODE	NAME
0x000000BC	NETWORK_BOOT_DUPLICATE_ADDRESS
0x000000BD	INVALID_HIBERNATED_STATE
0x000000BE	ATTEMPTED_WRITE_TO_READONLY_MEMORY
0x000000BF	MUTEX_ALREADY_OWNED
0x000000C1	SPECIAL_POOL_DETECTED_MEMORY_CORRUPTION
0x000000C2	BAD_POOL_CALLER
0x000000C4	DRIVER_VERIFIER_DETECTED_VIOLATION
0x000000C5	DRIVER_CORRUPTED_EXPOOL
0x000000C6	DRIVER_CAUGHT_MODIFYING_FREED_POOL
0x000000C7	TIMER_OR_DPC_INVALID
0x000000C8	IRQL_UNEXPECTED_VALUE
0x000000C9	DRIVER_VERIFIER_IOMANAGER_VIOLATION
0x000000CA	PNP_DETECTED_FATAL_ERROR
0x000000CB	DRIVER_LEFT_LOCKED_PAGES_IN_PROCESS
0x000000CC	PAGE_FAULT_IN_FREED_SPECIAL_POOL
0x000000CD	PAGE_FAULT_BEYOND_END_OF_ALLOCATION
0x000000CE	DRIVER_UNLOADED_WITHOUT_CANCELLING_PENDING_OPERATIONS
0x000000CF	TERMINAL_SERVER_DRIVER_MADE_INCORRECT_MEMORY_REFERENCE
0x000000D0	DRIVER_CORRUPTED_MMPOOL
0x000000D1	DRIVER_IRQL_NOT_LESS_OR_EQUAL
0x000000D2	BUGCODE_ID_DRIVER
0x000000D3	DRIVER_PORTION_MUST_BE_NONPAGED
0x000000D4	SYSTEM_SCAN_AT_RAISED_IRQL_CAUGHT_IMPROPER_DRIVER_UNLOAD
0x000000D5	DRIVER_PAGE_FAULT_IN_FREED_SPECIAL_POOL

CODE	NAME
0x000000D6	DRIVER_PAGE_FAULT_BEYOND_END_OF_ALLOCATION
0x000000D7	DRIVER_UNMAPPING_INVALID_VIEW
0x000000D8	DRIVER_USED_EXCESSIVE_PTES
0x000000D9	LOCKED_PAGES_TRACKER_CORRUPTION
0x000000DA	SYSTEM_PTE_MISUSE
0x000000DB	DRIVER_CORRUPTED_SYSPTES
0x000000DC	DRIVER_INVALID_STACK_ACCESS
0x000000DE	POOL_CORRUPTION_IN_FILE_AREA
0x000000DF	IMPERSONATING_WORKER_THREAD
0x000000E0	ACPI_BIOS_FATAL_ERROR
0x000000E1	WORKER_THREAD_RETURNED_AT_BAD_IRQL
0x000000E2	MANUALLY_INITIATED_CRASH
0x000000E3	RESOURCE_NOT_OWNED
0x000000E4	WORKER_INVALID
0x000000E6	DRIVER_VERIFIER_DMA_VIOLATION
0x000000E7	INVALID_FLOATING_POINT_STATE
0x000000E8	INVALID_CANCEL_OF_FILE_OPEN
0x000000E9	ACTIVE_EX_WORKER_THREAD_TERMINATION
0x000000EA	THREAD_STUCK_IN_DEVICE_DRIVER
0x000000EB	DIRTY_MAPPED_PAGES_CONGESTION
0x000000EC	SESSION_HAS_VALID_SPECIAL_POOL_ON_EXIT
0x000000ED	UNMOUNTABLE_BOOT_VOLUME
0x000000EF	CRITICAL_PROCESS_DIED
0x000000F0	STORAGE_MINIPORT_ERROR
0x000000F1	SCSI_VERIFIER_DETECTED_VIOLATION

CODE	NAME
0x000000F2	HARDWARE_INTERRUPT_STORM
0x000000F3	DISORDERLY_SHUTDOWN
0x000000F4	CRITICAL_OBJECT_TERMINATION
0x000000F5	FLTMGR_FILE_SYSTEM
0x000000F6	PCI_VERIFIER_DETECTED_VIOLATION
0x000000F7	DRIVER_OVERRAN_STACK_BUFFER
0x000000F8	RAMDISK_BOOT_INITIALIZATION_FAILED
0x000000F9	DRIVER_RETURNED_STATUS_REPARSE_FOR_VOLUME_OPEN
0x000000FA	HTTP_DRIVER_CORRUPTED
0x000000FC	ATTEMPTED_EXECUTE_OF_NOEXECUTE_MEMORY
0x000000FD	DIRTY_NOWRITE_PAGES_CONGESTION
0x000000FE	BUGCODE_USB_DRIVER
0x000000FF	RESERVE_QUEUE_OVERFLOW
0x00000100	LOADER_BLOCK_MISMATCH
0x00000101	CLOCK_WATCHDOG_TIMEOUT
0x00000102	DPC_WATCHDOG_TIMEOUT
0x00000103	MUP_FILE_SYSTEM
0x00000104	AGP_INVALID_ACCESS
0x00000105	AGP_GART_CORRUPTION
0x00000106	AGP_ILLEGALLY_REPROGRAMMED
0x00000108	THIRD_PARTY_FILE_SYSTEM_FAILURE
0x00000109	CRITICAL_STRUCTURE_CORRUPTION
0x0000010A	APP_TAGGING_INITIALIZATION_FAILED
0x0000010C	FSRTL_EXTRA_CREATE_PARAMETER_VIOLATION
0x0000010D	WDF_VIOLATION

CODE	NAME
0x0000010E	VIDEO_MEMORY_MANAGEMENT_INTERNAL
0x0000010F	RESOURCE_MANAGER_EXCEPTION_NOT_HANDLED
0x00000111	RECURSIVE_NMI
0x00000112	MSRPC_STATE_VIOLATION
0x00000113	VIDEO_DXGKRNL_FATAL_ERROR
0x00000114	VIDEO_SHADOW_DRIVER_FATAL_ERROR
0x00000115	AGP_INTERNAL
0x00000116	VIDEO_TDR_FAILURE
0x00000117	VIDEO_TDR_TIMEOUT_DETECTED
0x00000119	VIDEO_SCHEDULER_INTERNAL_ERROR
0x0000011A	EM_INITIALIZATION_FAILURE
0x0000011B	DRIVER_RETURNED_HOLDING_CANCEL_LOCK
0x0000011C	ATTEMPTED_WRITE_TO_CM_PROTECTED_STORAGE
0x0000011D	EVENT_TRACING_FATAL_ERROR
0x0000011E	TOO_MANY_RECURSIVE_FAULTS
0x0000011F	INVALID_DRIVER_HANDLE
0x00000120	BITLOCKER_FATAL_ERROR
0x00000121	DRIVER_VIOLATION
0x00000122	WHEA_INTERNAL_ERROR
0x00000123	CRYPTO_SELF_TEST_FAILURE
0x00000124	WHEA_UNCORRECTABLE_ERROR
0x00000125	NMR_INVALID_STATE
0x00000126	NETIO_INVALID_POOL_CALLER
0x00000127	PAGE_NOT_ZERO
0x00000128	WORKER_THREAD_RETURNED_WITH_BAD_IO_PRIORITY

CODE	NAME
0x00000129	WORKER_THREAD_RETURNED_WITH_BAD_PAGING_IO_PRIORITY
0x0000012A	MUI_NO_VALID_SYSTEM_LANGUAGE
0x0000012B	FAULTY_HARDWARE_CORRUPTED_PAGE
0x0000012C	EXFAT_FILE_SYSTEM
0x0000012D	VOLSNAP_OVERLAPPED_TABLE_ACCESS
0x0000012E	INVALID_MDL_RANGE
0x0000012F	VHD_BOOT_INITIALIZATION_FAILED
0x00000130	DYNAMIC_ADD_PROCESSOR_MISMATCH
0x00000131	INVALID_EXTENDED_PROCESSOR_STATE
0x00000132	RESOURCE_OWNER_POINTER_INVALID
0x00000133	DPC_WATCHDOG_VIOLATION
0x00000134	DRIVE_EXTENDER
0x00000135	REGISTRY_FILTER_DRIVER_EXCEPTION
0x00000136	VHD_BOOT_HOST_VOLUME_NOT_ENOUGH_SPACE
0x00000137	WIN32K_HANDLE_MANAGER
0x00000138	GPIO_CONTROLLER_DRIVER_ERROR
0x00000139	KERNEL_SECURITY_CHECK_FAILURE
0x0000013A	KERNEL_MODE_HEAP_CORRUPTION
0x0000013B	PASSIVE_INTERRUPT_ERROR
0x0000013C	INVALID_IO_BOOST_STATE
0x0000013D	CRITICAL_INITIALIZATION_FAILURE
0x00000140	STORAGE_DEVICE_ABNORMALITY_DETECTED
0x00000141	VIDEO_ENGINE_TIMEOUT_DETECTED
0x00000142	VIDEO_TDR_APPLICATION_BLOCKED
0x00000143	PROCESSOR_DRIVER_INTERNAL

CODE	NAME
0x00000144	BUGCODE_USB3_DRIVER
0x00000145	SECURE_BOOT_VIOLATION
0x00000147	ABNORMAL_RESET_DETECTED
0x00000149	REFS_FILE_SYSTEM
0x0000014A	KERNEL_WMI_INTERNAL
0x0000014B	SOC_SUBSYSTEM_FAILURE
0x0000014C	FATAL_ABNORMAL_RESET_ERROR
0x0000014D	EXCEPTION_SCOPE_INVALID
0x0000014E	SOC_CRITICAL_DEVICE_REMOVED
0x0000014F	PDC_WATCHDOG_TIMEOUT
0x00000150	TCPIP_AOAC_NIC_ACTIVE_REFERENCE_LEAK
0x00000151	UNSUPPORTED_INSTRUCTION_MODE
0x00000152	INVALID_PUSH_LOCK_FLAGS
0x00000153	KERNEL_LOCK_ENTRY_LEAKED_ON_THREAD_TERMINATION
0x00000154	UNEXPECTED_STORE_EXCEPTION
0x00000155	OS_DATA_TAMPERING
0x00000156	WINSOCK_DETECTED_HUNG_CLOSESOCKET_LIVEDUMP
0x00000157	KERNEL_THREAD_PRIORITY_FLOOR_VIOLATION
0x00000158	ILLEGAL_IOMMU_PAGE_FAULT
0x00000159	HAL_ILLEGAL_IOMMU_PAGE_FAULT
0x0000015A	SDBUS_INTERNAL_ERROR
0x0000015B	WORKER_THREAD_RETURNED_WITH_SYSTEM_PAGE_PRIORITY_ACTIVE
0x0000015C	PDC_WATCHDOG_TIMEOUT_LIVEDUMP
0x0000015D	SOC_SUBSYSTEM_FAILURE_LIVEDUMP

CODE	NAME
0x0000015E	BUGCODE_NDIS_DRIVER_LIVE_DUMP
0x0000015F	CONNECTED_STANDBY_WATCHDOG_TIMEOUT_LIVEDUMP
0x00000160	WIN32K_ATOMIC_CHECK_FAILURE
0x00000161	LIVE_SYSTEM_DUMP
0x00000162	KERNEL_AUTO_BOOST_INVALID_LOCK_RELEASE
0x00000163	WORKER_THREAD_TEST_CONDITION
0x00000164	WIN32K_CRITICAL_FAILURE
0x00000165	CLUSTER_CSV_STATUS_IO_TIMEOUT_LIVEDUMP
0x00000166	CLUSTER_RESOURCE_CALL_TIMEOUT_LIVEDUMP
0x00000167	CLUSTER_CSV_SNAPSHOT_DEVICE_INFO_TIMEOUT_LIVEDUMP
0x00000168	CLUSTER_CSV_STATE_TRANSITION_TIMEOUT_LIVEDUMP
0x00000169	CLUSTER_CSV_VOLUME_ARRIVAL_LIVEDUMP
0x0000016A	CLUSTER_CSV_VOLUME_REMOVAL_LIVEDUMP
0x0000016B	CLUSTER_CSV_CLUSTER_WATCHDOG_LIVEDUMP
0x0000016C	INVALID_RUNDOWN_PROTECTION_FLAGS
0x0000016D	INVALID_SLOT_ALLOCATOR_FLAGS
0x0000016E	ERESOURCE_INVALID_RELEASE
0x0000016F	CLUSTER_CSV_STATE_TRANSITION_INTERVAL_TIMEOUT_LIVEDUMP
0x00000170	CRYPTO_LIBRARY_INTERNAL_ERROR
0x00000171	CLUSTER_CSV_CLUSSVC_DISCONNECT_WATCHDOG
0x00000173	COREMSGCALL_INTERNAL_ERROR
0x00000174	COREMSG_INTERNAL_ERROR
0x00000175	PREVIOUS_FATAL_ABNORMAL_RESET_ERROR
0x00000178	ELAM_DRIVER_DETECTED_FATAL_ERROR

CODE	NAME
0x00000179	CLUSTER_CLUSPORT_STATUS_IO_TIMEOUT_LIVEDUMP
0x0000017B	PROFILER_CONFIGURATION_ILLEGAL
0x0000017C	PDC_LOCK_WATCHDOG_LIVEDUMP
0x0000017D	PDC_UNEXPECTED_REVOCATION_LIVEDUMP
0x0000017E	MICROCODE_REVISION_MISMATCH
0x00000187	VIDEO_DWMINIT_TIMEOUT_FALLBACK_BDD
0x00000188	CLUSTER_CSVFS_LIVEDUMP
0x00000189	BAD_OBJECT_HEADER
0x0000018B	SECURE_KERNEL_ERROR
0x0000018C	HYPERGUARD_VIOLATION
0x0000018D	SECURE_FAULT_UNHANDLED
0x0000018E	KERNEL_PARTITION_REFERENCE_VIOLATION
0x00000190	WIN32K_CRITICAL_FAILURE_LIVEDUMP
0x00000191	PF_DETECTED_CORRUPTION
0x00000192	KERNEL_AUTO_BOOST_LOCK_ACQUISITION_WITH_RAISED_IRQL
0x00000193	VIDEO_DXGKRNL_LIVEDUMP
0x00000195	SMB_SERVER_LIVEDUMP
0x00000196	LOADER_ROLLBACK_DETECTED
0x00000197	WIN32K_SECURITY_FAILURE
0x00000198	UFX_LIVEDUMP
0x00000199	KERNEL_STORAGE_SLOT_IN_USE
0x0000019A	WORKER_THREAD_RETURNED_WHILE_ATTACHED_TO_SILO
0x0000019B	TTM_FATAL_ERROR
0x0000019C	WIN32K_POWER_WATCHDOG_TIMEOUT

CODE	NAME
0x0000019D	CLUSTER_SVHDX_LIVEDUMP
0x000001A0	TTM_WATCHDOG_TIMEOUT
0x000001A3	CALL_HAS_NOT_RETURNED_WATCHDOG_TIMEOUT_LIVEDUMP
0x000001A4	DRIPS_SW_HW_DIVERGENCE_LIVEDUMP
0x000001A5	USB_DRIPS_BLOCKER_SURPRISE_REMOVAL_LIVEDUMP
0x000001A6	BLUETOOTH_ERROR_RECOVERY_LIVEDUMP
0x000001A7	SMB_REDIRECTOR_LIVEDUMP
0x000001A8	VIDEO_DXGKRNL_BLACK_SCREEN_LIVEDUMP
0x000001B0	VIDEO_MINIPORT_FAILED_LIVEDUMP
0x000001C4	DRIVER_VERIFIER_DETECTED_VIOLATION_LIVEDUMP
0x000001C5	IO_THREADPOOL_DEADLOCK_LIVEDUMP
0x000001C6	FAST_ERESOURCE_PRECONDITION_VIOLATION
0x000001C7	STORE_DATA_STRUCTURE_CORRUPTION
0x000001C8	MANUALLY_INITIATED_POWER_BUTTON_HOLD
0x000001C9	USER_MODE_HEALTH_MONITOR_LIVEDUMP
0x000001CA	SYNTHETIC_WATCHDOG_TIMEOUT
0x000001CB	INVALID_SILO_DETACH
0x000001CC	EXRESOURCE_TIMEOUT_LIVEDUMP
0x000001CD	INVALID_CALLBACK_STACK_ADDRESS
0x000001CE	INVALID_KERNEL_STACK_ADDRESS
0x000001CF	HARDWARE_WATCHDOG_TIMEOUT
0x000001D0	CPI_FIRMWARE_WATCHDOG_TIMEOUT
0x000001D1	TELEMETRY_ASSERTS_LIVEDUMP
0x000001D2	WORKER_THREAD_INVALID_STATE

CODE	NAME
0x000001D3	WFP_INVALID_OPERATION
0x000001D4	UCMUCSI_LIVEDUMP
0x000001D5	DRIVER_PNP_WATCHDOG
0x000001D6	WORKER_THREAD_RETURNED_WITH_NON_DEFAULT_WORKLOAD_CLASS
0x000001D7	EFS_FATAL_ERROR
0x000001D8	UCMUCSI_FAILURE
0x000001D9	HAL_IOMMU_INTERNAL_ERROR
0x000001DB	IPI_WATCHDOG_TIMEOUT
0x000001DC	DMA_COMMON_BUFFER_VECTOR_ERROR
0x00000356	XBOX_ERACTRL_CS_TIMEOUT
0x00000BFE	BC_BLUETOOTH_VERIFIER_FAULT
0x00000BFF	BC_BTHMINI_VERIFIER_FAULT
0x00020001	HYPERVISOR_ERROR
0x1000007E	SYSTEM_THREAD_EXCEPTION_NOT_HANDLED_M
0x1000007F	UNEXPECTED_KERNEL_MODE_TRAP_M
0x1000008E	KERNEL_MODE_EXCEPTION_NOT_HANDLED_M
0x100000EA	THREAD_STUCK_IN_DEVICE_DRIVER_M
0x4000008A	THREAD_TERMINATE_HELD_MUTEX
0xC0000218	STATUS_CANNOT_LOAD_REGISTRY_FILE
0xC000021A	WINLOGON_FATAL_ERROR
0xC0000221	STATUS_IMAGE_CHECKSUM_MISMATCH
0xDEADDEAD	MANUALLY_INITIATED_CRASH1

Bug Check 0x1: APC_INDEX_MISMATCH

10/23/2019 • 2 minutes to read • Edit Online

The APC_INDEX_MISMATCH bug check has a value of 0x00000001. This indicates that there has been a mismatch in the asynchronous procedure calls (APC) state index.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

APC_INDEX_MISMATCH parameters

PARAMETER	DESCRIPTION
1	The address of the system function (system call) or worker routine.
2	The value of the current thread's **ApcStateIndex** field.
3	The value of current thread's **CombinedApcDisable** field. This field consists of two separate 16-bit fields: (*Thread* >**SpecialApcDisable** << 16) | *Thread* >**KernelApcDisable**.
4	Call type: • 0 - System call • 1 - Worker routine

Cause

The most common cause of this bug check is when a file system or driver has a mismatched sequence of calls to disable and re-enable APCs. The key data item is the *Thread* >**CombinedApcDisable** field. The **CombinedApcDisable** field consists of two separate 16-bit fields: **SpecialApcDisable** and **KernelApcDisable**. A negative value of either field indicates that a driver has disabled special or normal APCs (respectively) without re-enabling them. A positive value indicates that a driver has enabled special or normal APCs too many times.

Resolution

Debugging with WinDbg

The **!analyze** debugger extension displays information about the bug check and can be helpful in determining the root cause.

You can use the **!apc** extension to display the contents of one or more asynchronous procedure calls (APCs).

You can also set a breakpoint in the code that precedes this stop code and attempt to single-step forward into the faulting code.

For more information about using WinDbg, see Crash dump analysis using the Windows debuggers (WinDbg).

Debugging without WinDbg

If you are not equipped to use the Windows debugger to work on this problem, you can use some basic troubleshooting techniques.

- Check the System Log in Event Viewer for additional error messages that might help identify the device or driver that is causing this bug check.
- If a driver is identified in the bug check message, disable the driver or check with the manufacturer for driver updates.
- Confirm that any new hardware that is installed is compatible with the installed version of Windows. For example, you can get information about required hardware at Windows 10 specifications.

For additional general troubleshooting information, see Blue screen data.

Remarks

This bug check is the result of an internal error in the kernel. This error occurs on exit from a system call. A possible cause for this bug check is a file system or driver that has a mismatched sequence of system calls to enter or leave guarded or critical regions. For example, each call to **KeEnterCriticalRegion** must have a matching call to **KeLeaveCriticalRegion**.

If you are developing a driver, you can use Static Driver Verifier, a static analysis tool available in the Windows Driver Kit, to detect problems in your code before you ship your driver. Run Static Driver Verifier with the CriticalRegions rule to verify that your source code uses these system calls in correct sequence.

Bug Check 0x2: DEVICE_QUEUE_NOT_BUSY

7/2/2019 • 2 minutes to read • Edit Online

The DEVICE_QUEUE_NOT_BUSY bug check has a value of 0x00000002.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Resolution

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause.

Bug Check 0x3: INVALID_AFFINITY_SET

7/2/2019 • 2 minutes to read • Edit Online

The INVALID_AFFINITY_SET bug check has a value of 0x00000003.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Resolution

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause.

Bug Check 0x4: INVALID_DATA_ACCESS_TRAP

7/2/2019 • 2 minutes to read • Edit Online

The INVALID_DATA_ACCESS_TRAP bug check has a value of 0x00000004. It indicates an incorrect data access trap.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Resolution

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause.

Bug Check 0x5: INVALID_PROCESS_ATTACH_ATTEMPT

7/2/2019 • 2 minutes to read • Edit Online

The INVALID_PROCESS_ATTACH_ATTEMPT bug check has a value of 0x00000005. This generally indicates that the thread was attached to a process in a situation where that is not allowed. For example, this bug check could occur if **KeAttachProcess** was called when the thread was already attached to a process (which is illegal), or if the thread returned from certain function calls in an attached state (which is invalid),

This bug check appears very infrequently.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

INVALID_PROCESS_ATTACH_ATTEMPT Parameters

PARAMETER	DESCRIPTION
1	The pointer to the dispatcher object for the target process, or if the thread is already attached, the pointer to the object for the original process.
2	The pointer to the dispatcher object of the process that the current thread is currently attached to.
3	The value of the thread's APC state index.
4	A non-zero value indicates that a DPC is running on the current processor.

Remarks

This bug check can occur if the driver calls the **KeAttachProcess** function and the thread is already attached to another process. It is better to use the **KeStackAttachProcess** function. If the current thread was already attached to another process, the **KeStackAttachProcess** function saves the current APC state before it attaches the current thread to the new process.

Bug Check 0x6: INVALID_PROCESS_DETACH_ATTEMPT

10/23/2019 • 2 minutes to read • Edit Online

The INVALID_PROCESS_DETACH_ATTEMPT bug check has a value of 0x00000006.

This bug check appears very infrequently. This bug check can be caused by calling the **KeStackAttachProcess** routine and subsequently calling **KeUnstackDetachProcess** in the driver's implementation of the **PLOAD_IMAGE_NOTIFY_ROUTINE** callback function. The callback runs in a thread of the process in which the image loaded.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Bug Check 0x7: INVALID_SOFTWARE_INTERRUPT

7/2/2019 • 2 minutes to read • Edit Online

The INVALID_SOFTWARE_INTERRUPT bug check has a value of 0x00000007.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Bug Check 0x8: IRQL_NOT_DISPATCH_LEVEL

11/4/2019 • 2 minutes to read • Edit Online

The IRQL_NOT_DISPATCH_LEVEL bug check has a value of 0x00000008.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Resolution

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause.

Bug Check 0x9: IRQL_NOT_GREATER_OR_EQUAL

7/2/2019 • 2 minutes to read • Edit Online

The IRQL_NOT_GREATER_OR_EQUAL bug check has a value of 0x00000009.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Resolution

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause.

Bug Check 0xA: IRQL_NOT_LESS_OR_EQUAL

10/15/2019 • 5 minutes to read • Edit Online

The IRQL_NOT_LESS_OR_EQUAL bug check has a value of 0x0000000A. This indicates that Microsoft Windows or a kernel-mode driver accessed paged memory at an invalid address while at a raised interrupt request level (IRQL). This is typically the result of either a bad pointer or a pageability problem.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

IRQL_NOT_LESS_OR_EQUAL parameters

PARAMETER	DESCRIPTION
1	The virtual memory address that could not be accessed. Use **!pool** on this address to see whether it's paged pool. These commands, may also be useful in gathering information about the failure: **!pte**, **!address**, and **ln** (list nearest symbols).
2	IRQL at time of the fault. VALUES: • **2**: The IRQL was DISPATCH_LEVEL at the time of the fault.
3	Bit field that describes the operation that caused the fault. Note that bit 3 is only available on chipsets that support this level of reporting. **Bit 0 values:** • 0 - Read operation • 1 - Write operation **Bit 3 values:** • 0 - Not an execute operation • 1 - Execute operation **Bit 0 and Bit 3 combined values:** • 0x0 - Fault trying to READ from the address in parameter 1. • 0x1 - Fault trying to WRITE to the address in parameter 1. • 0x8 - Fault trying to EXECUTE code from the address in parameter 1. This value is usually caused by: • Calling a function that cannot be called at DISPATCH_LEVEL while at DISPATCH_LEVEL. • Forgetting to release a spinlock. • Marking code as pageable when it must be non-pageable (for example, because the code acquires a spinlock, or is called in a deferred procedure call).

PARAMETER	DESCRIPTION
4	The instruction pointer at the time of the fault. Use the **ln** (list nearest symbols) command on this address to see the name of the function.

Cause

This bug check is usually caused by kernel-mode device drivers that use improper addresses.

This bug check indicates that an attempt was made to access an invalid address while at a raised interrupt request level (IRQL). This is either a bad memory pointer or a pageability problem with the device driver code.

Following are some general guidelines that you can use to categorize the type of coding error that caused the bug check:

- If parameter 1 is less than 0x1000, the issue is likely a NULL pointer dereference.
- If **!pool** reports that parameter 1 is paged pool (or other types of pageable memory), then the IRQL is too high to access this data. Run at a lower IRQL, or allocate the data in the nonpaged pool.
- If parameter 3 indicates that this was an attempt to execute pageable code, then the IRQL is too high to call this function. Run at a lower IRQL, or do not mark the code as pageable.
- Otherwise, this may be a bad pointer, possibly caused by use-after-free or bit-flipping. Investigate the validity of parameter 1 with **!pte**, **!address**, and **ln** (list nearest symbols).

Resolution

If a kernel debugger is available, obtain a stack trace. Start by running the **!analyze** debugger extension to display information about the bug check. (The **!analyze** extension can be helpful in determining the root cause.) Next, enter one of the **k*** (display stack backtrace) commands to view the call stack.

Gather Information

Examine the name of the driver if that was listed on the blue screen.

Check the System Log in Event Viewer for additional error messages that might help pinpoint the device or driver that is causing the error. Look for critical errors in the system log that occurred in the same time window as the blue screen.

Driver Verifier

Driver Verifier is a tool that runs in real time to examine the behavior of drivers. For example, Driver Verifier checks the use of memory resources, such as memory pools. If it identifies errors in the execution of driver code, it proactively creates an exception to allow that part of the driver code to be further scrutinized. Driver Verifier Manager is built into Windows and is available on all Windows PCs.

To start Driver Verifier Manager, type **verifier** at a command prompt. You can configure which drivers to verify. The code that verifies drivers adds overhead as it runs, so try to verify the smallest number of drivers as possible. For more information, see Driver Verifier.

Following is a debugging example:

```
kd> .bugcheck       [Lists bug check data.]
Bugcheck code 0000000a
Arguments 00000000 0000001c 00000000 00000000

kd> kb [Lists the stack trace.]
ChildEBP RetAddr  Args to Child
8013ed5c 801263ba 00000000 00000000 e12ab000 NT!_DbgBreakPoint
8013eecc 801389ee 0000000a 00000000 0000001c NT!_KeBugCheckEx+0x194
8013eecc 00000000 0000000a 00000000 0000001c NT!_KiTrap0E+0x256
8013ed5c 801263ba 00000000 00000000 e12ab000
8013ef64 00000246 fe551aa1 ff690268 00000002 NT!_KeBugCheckEx+0x194

kd> kv [Lists the trap frames.]
ChildEBP RetAddr  Args to Child
8013ed5c 801263ba 00000000 00000000 e12ab000 NT!_DbgBreakPoint (FPO: [0,0,0])
8013eecc 801389ee 0000000a 00000000 0000001c NT!_KeBugCheckEx+0x194
8013eecc 00000000 0000000a 00000000 0000001c NT!_KiTrap0E+0x256 (FPO: [0,0] TrapFrame @ 8013eee8)
8013ed5c 801263ba 00000000 00000000 e12ab000
8013ef64 00000246 fe551aa1 ff690268 00000002 NT!_KeBugCheckEx+0x194

kd> .trap 8013eee8 [Gets the registers for the trap frame at the time of the fault.]
eax=dec80201 ebx=ffdff420 ecx=8013c71c edx=000003f8 esi=00000000 edi=87038e10
eip=00000000 esp=8013ef5c ebp=8013ef64 iopl=0         nv up ei pl nz na pe nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00010202
ErrCode = 00000000
00000000 ??????????????    [The current instruction pointer is NULL.]

kd> kb       [Gives the stack trace before the fault.]
ChildEBP RetAddr  Args to Child
8013ef68 fe551aa1 ff690268 00000002 fe5620d2 NT!_DbgBreakPoint
8013ef74 fe5620d2 fe5620da ff690268 80404690
NDIS!_EthFilterIndicateReceiveComplete+0x31
8013ef64 00000246 fe551aa1 ff690268 00000002 elnkii!_ElnkiiRcvInterruptDpc+0x1d0
```

Remarks

The error that generates this bug check usually occurs after the installation of a faulty device driver, system service, or BIOS.

If you encounter bug check 0xA while upgrading to a newer version of Windows, the error might be caused by a device driver, a system service, a virus scanner, or a backup tool that is incompatible with the new version.

Resolving a faulty hardware problem: If hardware has been added to the system recently, remove it to see if the error recurs. If existing hardware has failed, remove or replace the faulty component. Run hardware diagnostics that are supplied by the system manufacturer. For details on these procedures, see the owner's manual for your computer.

Resolving a faulty system service problem: Disable the service and confirm whether doing so resolves the error. If so, contact the manufacturer of the system service about a possible update. If the error occurs during system startup, investigate the Windows repair options. For more information, see Recovery options in Windows 10.

Resolving an antivirus software problem: Disable the program and confirm whether doing so resolves the error. If it does, contact the manufacturer of the program about a possible update.

For general information about troubleshooting bug checks, see Blue screen data.

Bug Check 0xB: NO_EXCEPTION_HANDLING_SUPPORT

7/2/2019 • 2 minutes to read • Edit Online

The NO_EXCEPTION_HANDLING_SUPPORT bug check has a value of 0x0000000B.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Bug Check 0xC: MAXIMUM_WAIT_OBJECTS_EXCEEDED

7/2/2019 • 2 minutes to read • Edit Online

The MAXIMUM_WAIT_OBJECTS_EXCEEDED bug check has a value of 0x0000000C. This indicates that the current thread exceeded the permitted number of wait objects.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

MAXIMUM_WAIT_OBJECTS_EXCEEDED Parameters

None

Cause

This bug check results from the improper use of **KeWaitForMultipleObjects** or **FsRtlCancellableWaitForMultipleObjects**.

The caller may pass a pointer to a buffer in this routine's *WaitBlockArray* parameter. The system will use this buffer to keep track of wait objects.

If a buffer is supplied, the *Count* parameter may not exceed MAXIMUM_WAIT_OBJECTS. If no buffer is supplied, the *Count* parameter may not exceed THREAD_WAIT_OBJECTS.

If the value of *Count* exceeds the allowable value, this bug check is issued.

Bug Check 0xD: MUTEX_LEVEL_NUMBER_VIOLATION

7/2/2019 • 2 minutes to read • Edit Online

The MUTEX_LEVEL_NUMBER_VIOLATION bug check has a value of 0x0000000D.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Bug Check 0xE: NO_USER_MODE_CONTEXT

11/4/2019 • 2 minutes to read • Edit Online

The NO_USER_MODE_CONTEXT bug check has a value of 0x0000000E.

In the process of starting a system thread, if control returns from the initial thread procedure, a bug check will occur.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Resolution

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause.

Bug Check 0xF: SPIN_LOCK_ALREADY_OWNED

7/2/2019 • 2 minutes to read • Edit Online

The SPIN_LOCK_ALREADY_OWNED bug check has a value of 0x0000000F. This indicates that a request for a spin lock has been initiated when the spin lock was already owned.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

SPIN_LOCK_ALREADY_OWNED Parameters

None

Cause

Typically, this error is caused by a recursive request for a spin lock. It can also occur if something similar to a recursive request for a spin lock has been initiated--for example, when a spin lock has been acquired by a thread, and then that same thread calls a function, which also tries to acquire a spin lock. The second attempt to acquire a spin lock is not blocked in this case because doing so would result in an unrecoverable deadlock. If the calls are made on more than one processor, then one processor will be blocked until the other processor releases the lock.

This error can also occur, without explicit recursion, when all threads and all spin locks are assigned an IRQL. Spin lock IRQLs are always greater than or equal to DPC level, but this is not true for threads. However, a thread that is holding a spin lock must maintain an IRQL greater than or equal to that of the spin lock. Decreasing the thread IRQL below the IRQL level of the spin lock that it is holding allows another thread to be scheduled on the processor. This new thread could then attempt to acquire the same spin lock.

Resolution

Ensure that you are not recursively acquiring the lock. And, for threads that hold a spin lock, ensure that you are not decreasing the thread IRQL to a level below the IRQL of the spin lock that it is holding.

Bug Check 0x10: SPIN_LOCK_NOT_OWNED

7/2/2019 • 2 minutes to read • Edit Online

The SPIN_LOCK_NOT_OWNED bug check has a value of 0x00000010.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Bug Check 0x11: THREAD_NOT_MUTEX_OWNER

7/2/2019 • 2 minutes to read • Edit Online

The THREAD_NOT_MUTEX_OWNER bug check has a value of 0x00000011.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Bug Check 0x12: TRAP_CAUSE_UNKNOWN

7/2/2019 • 2 minutes to read • Edit Online

The TRAP_CAUSE_UNKNOWN bug check has a value of 0x00000012. This indicates that an unknown exception has occurred.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

TRAP_CAUSE_UNKNOWN Parameters

PARAMETER	DESCRIPTION
1	Type of TRAP_CAUSE_UNKNOWN **VALUES** 1 - Unexpected interrupt. (Parameter 2 – Interrupt Vector) 2 - Unknown floating point exception. 3 - The enabled and asserted status bits (see processor definition).
2	Dependent on Arg1
3	Reserved
4	Reserved

Resolution

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause.

To start, examine the stack trace using the **k, kb, kc, kd, kp, kP, kv (Display Stack Backtrace)** command. You can specify the processor number to examine the stacks on all processors.

You can also set a breakpoint in the code leading up to this stop code and attempt to single step forward into the faulting code.

The !idt extension can be used to display the interrupt service routines (ISRs) for a specified interrupt dispatch table (IDT).

Some of the techniques described in Debugging an Interrupt Storm can be used with the unexpected interrupts.

For general information about working with crash dumps, see Crash dump analysis using the Windows debuggers (WinDbg).

If you are not equipped to use the Windows debugger to work on this problem, you can use some basic troubleshooting techniques.

- Check the System Log in Event Viewer for additional error messages that might help identify the device or driver that is causing this bug check.
- If a driver is identified in the bug check message, disable the driver or check with the manufacturer for driver updates.
- Confirm that any new hardware that is installed is compatible with the installed version of Windows. For example, you can get information about required hardware at Windows 10 Specifications.
- For additional general troubleshooting information, see **Blue Screen Data**.

Bug Check 0x13: EMPTY_THREAD_REAPER_LIST

7/2/2019 • 2 minutes to read • Edit Online

The EMPTY_THREAD_REAPER_LIST bug check has a value of 0x00000013.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Bug Check 0x14: CREATE_DELETE_LOCK_NOT_LOCKED

7/2/2019 • 2 minutes to read • Edit Online

The CREATE_DELETE_LOCK_NOT_LOCKED bug check has a value of 0x00000014.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Bug Check 0x15: LAST_CHANCE_CALLED_FROM_KMODE

7/2/2019 • 2 minutes to read • Edit Online

The LAST_CHANCE_CALLED_FROM_KMODE bug check has a value of 0x00000015.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Bug Check 0x16: CID_HANDLE_CREATION

7/2/2019 • 2 minutes to read • Edit Online

The CID_HANDLE_CREATION bug check has a value of 0x00000016.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Bug Check 0x17: CID_HANDLE_DELETION

7/2/2019 • 2 minutes to read • Edit Online

The CID_HANDLE_DELETION bug check has a value of 0x00000017.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Bug Check 0x18: REFERENCE_BY_POINTER

7/2/2019 • 2 minutes to read • Edit Online

The REFERENCE_BY_POINTER bug check has a value of 0x00000018. This indicates that the reference count of an object is illegal for the current state of the object.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

REFERENCE_BY_POINTER Parameters

PARAMETER	DESCRIPTION
1	Object type of the object whose reference count is being lowered.
2	Object whose reference count is being lowered.
3	Reserved
4	Reserved

Cause

The reference count of an object is illegal for the current state of the object. Each time a driver uses a pointer to an object, the driver calls a kernel routine to increase the reference count of the object by one. When the driver is done with the pointer, the driver calls another kernel routine to decrease the reference count by one.

Drivers must match calls to the routines that increase (*reference*) and decrease (*dereference*) the reference count. This bug check is caused by an inconsistency in the object's reference count. Typically, the inconsistency is caused by a driver that decreases the reference count of an object too many times, making extra calls that dereference the object. This bug check can occur because an object's reference count goes to zero while there are still open handles to the object. It might also occur when the object's reference count drops below zero, whether or not there are open handles to the object.

Resolution

Make sure that the driver matches calls to the routines that increase and decrease the reference count of the object. Make sure that your driver does not make extra calls to routines that dereference the object (see Parameter 2).

You can use a debugger to help analyze this problem. For more information, see Crash dump analysis using the Windows debuggers (WinDbg). The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause.

To find the handle and pointer count on the object, use the **!object** debugger command.

kd> !object *address*

Where *address* is the address of the object given in Parameter 2.

You can also set a breakpoint in the code leading up to this stop code and attempt to single step forward into the faulting code.

If you are not equipped to use the Windows debugger to work on this problem, you can use some basic troubleshooting techniques.

- Check the System Log in Event Viewer for additional error messages that might help identify the device or driver that is causing this bug check.
- If a driver is identified in the bug check message, disable the driver or check with the manufacturer for driver updates.
- Confirm that any new hardware that is installed is compatible with the installed version of Windows. For example, you can get information about required hardware at Windows 10 Specifications.
- For additional general troubleshooting information, see **Blue Screen Data**.

(Developer Content) Bug Check 0x19: BAD_POOL_HEADER

7/2/2019 • 3 minutes to read • Edit Online

The BAD_POOL_HEADER bug check has a value of 0x00000019. This indicates that a pool header is corrupt.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

BAD_POOL_HEADER Parameters

Parameter 1 indicates the type of violation. The meaning of the other parameters depends on the value of Parameter 1.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x2	The pool entry being checked	The size of the pool block	0	The special pool pattern check failed. (The owner has likely corrupted the pool block.)
0x3	The pool entry being checked	The read-back **flink** freelist value	The read-back **blink** freelist value	The pool freelist is corrupt. (In a healthy list, the values of Parameters 2, 3, and 4 should be identical.)
0x5	One of the pool entries	Reserved	The other pool entry	A pair of adjacent pool entries have headers that contradict each other. At least one of them is corrupt.
0x6	One incorrectly-calculated entry	Reserved	The bad entry that caused the miscalculation	The pool block header's previous size is too large.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x7	0	Reserved	The bad pool entry	The pool block header size is corrupt.
0x8	0	Reserved	The bad pool entry	The pool block header size is zero.
0x9	One incorrectly-calculated entry	Reserved	The bad entry that caused the miscalculation	The pool block header size is corrupted (it is too large).
0xA	The pool entry that should have been found	Reserved	The virtual address of the page that should have contained the pool entry	The pool block header size is corrupt.
0xD, 0xE, 0xF, 0x23, 0x24, 0x25	Reserved	Reserved	Reserved	The pool header of a freed block has been modified after it was freed. This is not typically the fault of the prior owner of the freed block; instead it is usually (but not always) due to the block preceding the freed block being overrun.
0x20	The pool entry that should have been found	The next pool entry	Reserved	The pool block header size is corrupt.
0X21	The pool pointer being freed	The number of bytes allocated for the pool block	The corrupted value found following the pool block	The data following the pool block being freed is corrupt. Typically this means the consumer (call stack) has overrun the block.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0X22	The address being freed	Reserved	Reserved	An address being freed does not have a tracking entry. This is usually because the call stack is trying to free a pointer that either has already been freed or was never allocated to begin with.

Cause

The pool is already corrupted at the time of the current request.

This may or may not be due to the caller.

Resolution

The internal pool links must be walked using the kernel debugger to figure out a possible cause of the problem.

Then you can use special pool for the suspect pool tags, or use Driver Verifier "Special Pool" option on the suspect driver. The **!analyze** extension may be of help in pinpointing the suspect driver, but this is frequently not the case with pool corrupters.

Use the steps described in **Blue Screen Data** to gather the Stop Code Parameters. Use the stop code parameters to determine the specific type of code behavior you are working to track down.

Driver Verifier

Driver Verifier is a tool that runs in real time to examine the behavior of drivers. If it see errors in the execution of driver code, it proactively creates an exception to allow that part of the driver code to be further scrutinized. The driver verifier manager is built into Windows and is available on all Windows PCs. To start the driver verifier manager, type *Verifier* at a command prompt. You can configure which drivers you would like to verify. The code that verifies drivers adds overhead as it runs, so try and verify the smallest number of drivers as possible. For more information, see Driver Verifier.

Windows Memory Diagnostics

If this Bug Check appears inconsistently, it could be related to faulty physical memory.

Run the Windows Memory Diagnostics tool, to test the memory. In the control panel search box, type Memory, and then click **Diagnose your computer's memory problems**. After the test is run, use Event viewer to view the results under the System log. Look for the *MemoryDiagnostics-Results* entry to view the results.

Bug Check 0x1A: MEMORY_MANAGEMENT

7/2/2019 • 6 minutes to read • Edit Online

The MEMORY_MANAGEMENT bug check has a value of 0x0000001A. This indicates that a severe memory management error occurred.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

MEMORY_MANAGEMENT Parameters

Parameter 1 identifies the exact violation.

PARAMETER 1	CAUSE OF ERROR
0x1	The fork clone block reference count is corrupt. (This only occurs on checked builds of Windows.)
0x31	The image relocation fix-up table or code stream has been corrupted. This is probably a hardware error.
0x3f	An inpage operation failed with a CRC error. Parameter 2 contains the pagefile offset. Parameter 3 contains the page CRC value. Parameter 4 contains the expected CRC value.
0x403	The page table and PFNs are out of sync . This is probably a hardware error, especially if parameters 3 & 4 differ by only a single bit.
0x404	In the process of deleting a system page there was an inconsistency between the Page Frame Number (PFN) and the current Page Table Entry (PTE) pointer. Parameter 2 is the expected PTE. Parameter 3 is the PTE contents and parameter 4 is the PFN's PTE.
0x411	A page table entry (PTE) has been corrupted. Parameter 2 is the address of the PTE.
0x777	The caller is unlocking a system cache address that is not currently locked. (This address was either never mapped or is being unlocked twice.)

PARAMETER 1	CAUSE OF ERROR
0x778	The system is using the very last system cache view address, instead of preserving it.
0x780 0x781	The PTEs mapping the argument system cache view have been corrupted.
0x1000	A caller of **MmGetSystemAddressForMdl*** tried to map a fully-cached physical page as non-cached. This action would cause a conflicting hardware translation buffer entry, and so it was refused by the operating system. Since the caller specified "bug check on failure" in the requesting MDL, the system had no choice but to issue a bug check in this instance.
0x1010	The caller is unlocking a pageable section that is not currently locked. (This section was either never locked or is being unlocked twice.)
0x1233	A driver tried to map a physical memory page that was not locked. This is illegal because the contents or attributes of the page can change at any time. This is a bug in the code that made the mapping call. Parameter 2 is the page frame number of the physical page that the driver attempted to map.
0x1234	The caller is trying lock a nonexistent pageable section.
0x1235	The caller is trying to protect an MDL with an invalid mapping.
0x1236	The caller specified an MDL that contains an unlocked (or invalid) physical page. Parameter 2 contains a pointer to the MDL. Parameter 3 contains a pointer to the invalid PFN. Parameter 4 contains the invalid PFN value.
0x3300	In the process of performing a write, the referenced virtual address is mistakenly marked as copy on write. Parameter 2 is the FaultingAddress. Parameter 3 is the PTE contents. Parameter 4 indicates the virtual address space type.
0x3451	The PTEs of a kernel thread stack that has been swapped out are corrupted.
0x3453	All the page table pages of an exited process could not be deleted due to outstanding references. This typically indicates corruption in the process' page table structures.

PARAMETER 1	CAUSE OF ERROR
0x4477	A driver tried to write to an unallocated address in the user space of the system process. Parameter 2 contains the address of the attempted write.
0x5003	The working set free list is corrupt. This is probably a hardware error.
0x5100	The allocation bitmap is corrupt. The memory manager is about to overwrite a virtual address that was already in use.
0x5200	A page on a free pool SLIST has been corrupted. This can be the result of a write-after-free bug in a driver, or an overrun from a previous page. Parameter 2 contains the address of a free pool block. Parameter 4 contains the value that was expected to be at that address. Parameter 3 contains the actual value that was found.
0x6001	The memory store component's private memory range is corrupted, causing it to become inaccessible. Parameter 2 is the returned status. Parameter 3 is the virtual address in the store's private memory range. Parameter 4 is the MemoryDescriptorList.
0x8884	(Windows 7 only). Two pages on the standby list that were supposed to have identical page priority values do not, in fact, have identical page priority values. The differing values are captured in parameter 4.
0x8888 0x8889	Internal memory management structures are corrupted.
0x888A	Internal memory management structures (likely the PTE or PFN) are corrupted.
0x9696	A PFN (parameter 2) was encountered with a corrupted linkage no longer connected to its top level process. This indicates corruption in the PFN structures.
0x15001	An error occurred In the process of un-securing memory that was previously secured. This can happen when the caller mistakenly invoked MmUnsecureVirtualMemory in the wrong process context.

PARAMETER 1	CAUSE OF ERROR
0x41201	In the process of querying a virtual address, there was an inconsistency between the Page Frame Number(PFN) and the current Page Table Entry (PTE) pointer. Parameter 2 is the corresponding PTE. Parameter 3 is the PTE contents and parameter 4 is the virtual address descriptor.
0x41283	The working set index encoded in the PTE is corrupted.
0x41284	A PTE or the working set list is corrupted.
0x41286	The caller is trying to free an invalid pool address.
0x41785	The working set list is corrupted.
0x41287	An illegal page fault occurred while holding working set synchronization. Parameter 2 contains the referenced virtual address.
0x41790	A page table page has been corrupted. On a 64 bit version of Windows, parameter 2 contains the address of the PFN for the corrupted page table page. On a 32 bit version of Windows, parameter 2 contains a pointer to the number of used PTEs, and parameter 3 contains the number of used PTEs.
0x41792	A corrupted PTE has been detected. Parameter 2 contains the address of the PTE. Parameters 3/4 contain the low/high parts of the PTE.
0x41793	A page table page has been corrupted. Parameter 2 contains a pointer to the last processed PTE. Parameter 3 contains the number of non-zero PTEs found. Parameter 4 contains the expected number of non-zero PTEs in the page table. This memory parameter has been deprecated and is no longer available after Windows 10 version 1803.
0x61940	A PDE has been unexpectedly invalidated.
0x61941	The paging hierarchy is corrupt. Parameter 2 is a pointer to the virtual address which caused the fault.
0x61946	The MDL being created is flawed. This almost always means the driver calling **MmProbeAndLockPages** is at fault. Typically the driver is attempting to create a Write MDL when it is being asked to process a paging Read.

PARAMETER 1	CAUSE OF ERROR
0x61948	In the process of decrementing the reference counts for an I/O space region, its accounting node could not be found. Typically this means the argument range was never locked or has already been unlocked. Parameter 2 is the base I/O frame. Parameter 3 is the number of pages in the region, and parameter 4 is the specific I/O frame whose node could not be found.
0x61949	The IoPageFrameNode is null. Parameter 2 is PageFrameIndex.
0x6194A	An error occurred when decrementing the reference counts on I/O space physical pages which are being unmapped. An entry which is not currently referenced is being dereferenced. Parameter 2 and 3 describe the caller's I/O space range being unmapped, and parameter 4 is the I/O space physical page which is expected to be referenced but is not.
0x03030303	The boot loader is broken. (This value applies only to Intel Itanium machines.)
0x03030308	The range to remove (or truncate) is in use by the loader so it cannot be safely removed, so the system must issue a stop code. Parameter 2 is HighestPhysicalPage.

Resolution

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause.

Running the **Windows Memory Diagnostic** tool could be useful as well to exclude any kind of problem affecting the physical memory modules.

Bug Check 0x1B: PFN_SHARE_COUNT

7/2/2019 • 2 minutes to read • Edit Online

The PFN_SHARE_COUNT bug check has a value of 0x0000001B.

This bug check code is no longer used by the Windows operating system.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Bug Check 0x1C: PFN_REFERENCE_COUNT

7/2/2019 • 2 minutes to read • Edit Online

The PFN_REFERENCE_COUNT bug check has a value of 0x0000001C.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Resolution

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause.

Bug Check 0x1D: NO_SPIN_LOCK_AVAILABLE

7/2/2019 • 2 minutes to read • Edit Online

The NO_SPIN_LOCK_AVAILABLE bug check has a value of 0x0000001D.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Bug Check 0x1E: KMODE_EXCEPTION_NOT_HANDLED

10/15/2019 • 5 minutes to read • Edit Online

The KMODE_EXCEPTION_NOT_HANDLED bug check has a value of 0x0000001E. This indicates that a kernel-mode program generated an exception that the error handler did not catch.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

KMODE_EXCEPTION_NOT_HANDLED parameters

PARAMETER	DESCRIPTION
1	The exception code that was not handled.
2	The address at which the exception occurred.
3	Parameter 0 of the exception.
4	Parameter 1 of the exception.

Cause

To interpret this bug check, you must identify which exception was generated.

Common exception codes include:

- 0x80000002: STATUS_DATATYPE_MISALIGNMENT

  An unaligned data reference was encountered.

- 0x80000003: STATUS_BREAKPOINT

  A breakpoint or ASSERT was encountered when no kernel debugger was attached to the system.

- 0xC0000005: STATUS_ACCESS_VIOLATION

  A memory access violation occurred. (Parameter 4 of the bug check is the address that the driver attempted to access.)

For a complete list of exception codes, see NTSTATUS values. The exception codes are defined in *ntstatus.h*, a header file provided by the Windows Driver Kit. (For more info, see Header files in the Windows Driver Kit).

Remarks

If you are not equipped to debug this problem, you can use some basic troubleshooting techniques described in Blue screen data. If a driver is identified in the bug check message, disable the driver or check with the manufacturer for driver updates.

Hardware incompatibility

Confirm that any new hardware that is installed is compatible with the installed version of Windows. For example, you can get information about required hardware at Windows 10 Specifications.

Faulty device driver or system service

In addition, a faulty device driver or system service might be responsible for this error. Hardware issues, such as BIOS incompatibilities, memory conflicts, and IRQ conflicts can also generate this error.

If a driver is listed by name within the bug check message, disable or remove that driver. Disable or remove any drivers or services that were recently added. If the error occurs during the startup sequence, and the system partition is formatted with NTFS file system, you might be able to use Safe Mode to disable the driver in Device Manager.

Check the System Log in Event Viewer for additional error messages that might help pinpoint the device or driver that is causing bug check 0x1E. Also run hardware diagnostics, especially the memory scanner, supplied by the system manufacturer. For details on these procedures, see the owner's manual for your computer.

The error that generates this message can occur after the first restart during Windows Setup, or after Setup is finished. A possible cause of the error is a system BIOS incompatibility. BIOS problems can be resolved by upgrading the system BIOS version.

Resolution

If you plan to debug this problem, you may find it difficult to obtain a stack trace. The exception address (parameter 2) should pinpoint the driver or function that caused this problem.

Exception code 0x80000003 indicates that a hard-coded breakpoint or assertion was hit, but the system was started with the **/NODEBUG** switch. This problem should rarely occur. If it occurs repeatedly, make sure that a kernel debugger is connected and that the system is started with the **/DEBUG** switch.

If exception code 0x80000002 occurs, the trap frame will supply additional information.

Unknown cause

If the specific cause of the exception is unknown, consider using the following procedure to obtain a stack trace.

> **NOTE**
>
> This procedure assumes that you can locate **NT!PspUnhandledExceptionInSystemThread**. However, in some cases (such as an access violation crash) you will not be able to do this. In that case, look for **ntoskrnl!KiDispatchException**. The third parameter passed to this function is a trap frame address. Use the **.trap** (display trap frame) command with this address to set the Register Context to the proper value. You can then perform stack traces and issue other commands.

To get a stack trace if the normal stack tracing procedures fail

1. Use the **kb** (display stack backtrace) command to display parameters in the stack trace. Look for the call to **NT!PspUnhandledExceptionInSystemThread**. (If this function is not listed, see the note below.)
2. The first parameter to **NT!PspUnhandledExceptionInSystemThread** is a pointer to a structure, which contains pointers to an **except** statement:

```
typedef struct _EXCEPTION_POINTERS {
    PEXCEPTION_RECORD ExceptionRecord;
    PCONTEXT ContextRecord;
    } EXCEPTION_POINTERS, *PEXCEPTION_POINTERS;

ULONG PspUnhandledExceptionInSystemThread(
    IN PEXCEPTION_POINTERS ExceptionPointers
    )
```

Use the **dd** (display memory) command on that address to display the necessary data.

3. The first retrieved value is an exception record and the second is a context record. Use the **.exr** (display exception record) command and the **.cxr** (display context record) command with these two values as their arguments, respectively.

4. After the **.cxr** command executes, use the **kb** command to display a stack trace that is based on the context record information. This stack trace indicates the calling stack where the unhandled exception occurred.

Example bug check

Following is an example of bug check 0x1E on an x86 processor:

```
kd> .bugcheck                get the bug check data
Bugcheck code 0000001e
Arguments c0000005 8013cd0a 00000000 0362cffff

kd> kb                       start with a stack trace
FramePtr  RetAddr   Param1   Param2   Param3   Function Name
8013ed5c  801263ba  00000000 00000000 fe40cb00 NT!_DbgBreakPoint
8013eecc  8013313c  0000001e c0000005 8013cd0a NT!_KeBugCheckEx+0x194
fe40cad0  8013318e  fe40caf8 801359ff fe40cb00 NT!PspUnhandledExceptionInSystemThread+0x18
fe40cad8  801359ff  fe40cb00 00000000 fe40cb00 NT!PspSystemThreadStartup+0x4a
fe40cf7c  8013cb8e  fe43a44c ff6ce388 00000000 NT!_except_handler3+0x47
00000000  00000000  00000000 00000000 00000000 NT!KiThreadStartup+0xe

kd> dd fe40caf8 L2           dump EXCEPTION_POINTERS structure
0xFE40CAF8  fe40cd88 fe40cbc4                  ..@...@.

kd> .exr fe40cd88            first DWORD is the exception record
Exception Record @ FE40CD88:
   ExceptionCode: c0000005
  ExceptionFlags: 00000000
  Chained Record: 00000000
ExceptionAddress: 8013cd0a
NumberParameters: 00000002
   Parameter[0]: 00000000
   Parameter[1]: 0362cfff

kd> .cxr fe40cbc4            second DWORD is the context record
CtxFlags: 00010017
eax=00087000 ebx=00000000 ecx=03ff0000 edx=ff63d000 esi=0362cfff edi=036b3fff
eip=8013cd0a esp=fe40ce50 ebp=fe40cef8 iopl=0         nv dn ei pl nz ac po cy
vip=0    vif=0
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00010617
0x8013cd0a  f3a4             rep movsb

kd> kb                       kb gives stack for context record
ChildEBP RetAddr  Args to Child
fe40ce54 80402e09 ff6c4000 ff63d000 03ff0000 NT!_RtlMoveMemory@12+0x3e
fe40ce68 80403c18 ffbc0c28 ff6ce008 ff6c4000 HAL!_HalpCopyBufferMap@20+0x49
fe40ce9c fe43b1e4 ff6cef90 ffbc0c28 ff6ce009 HAL!_IoFlushAdapterBuffers@24+0x148
fe40ceb8 fe4385b4 ff6ce388 6cd00800 ffbc0c28 QIC117!_kdi_FlushDMABuffers@20+0x28
fe40cef8 fe439894 ff6cd008 ffb6c820 fe40cf4c QIC117!_cqd_CmdReadWrite@8+0x26e
fe40cf18 fe437d92 ff6cd008 ffb6c820 ff6e4e50 QIC117!_cqd_DispatchFRB@8+0x210
fe40cf30 fe43a4f5 ff6cd008 ffb6c820 00000000 QIC117!_cqd_ProcessFRB@8+0x134
fe40cf4c 80133184 ff6ce388 00000000 00000000 QIC117!_kdi_ThreadRun@4+0xa9
fe40cf7c 8013cb8e fe43a44c ff6ce388 00000000 NT!_PspSystemThreadStartup@8+0x40
```

Bug Check 0x1F: SHARED_RESOURCE_CONV_ERROR

7/2/2019 • 2 minutes to read • Edit Online

The SHARED_RESOURCE_CONV_ERROR bug check has a value of 0x0000001F.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Bug Check 0x20: KERNEL_APC_PENDING_DURING_EXIT

7/2/2019 • 2 minutes to read • Edit Online

The KERNEL_APC_PENDING_DURING_EXIT bug check has a value of 0x00000020. This indicates that an asynchronous procedure call (APC) was still pending when a thread exited.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

KERNEL_APC_PENDING_DURING_EXIT Parameters

PARAMETER	DESCRIPTION
1	The address of the APC found pending during exit
2	The thread's APC disable count
3	The current IRQL
4	Reserved

Cause

The key data item is the APC disable count (Parameter 2) for the thread. If the count is nonzero, it will indicate the source of the problem.

The APC disable count is decremented each time a driver calls **KeEnterCriticalRegion**, **FsRtlEnterFileSystem**, or acquires a mutex.

The APC disable count is incremented each time a driver calls **KeLeaveCriticalRegion**, **KeReleaseMutex**, or **FsRtlExitFileSystem**.

Because these calls should always be in pairs, the APC disable count should be zero when a thread exits. A negative value indicates that a driver has disabled APC calls without re-enabling them. A positive value indicates that the reverse is true.

If you ever see this error, be very suspicious of all drivers installed on the machine -- especially unusual or non-standard drivers.

This current IRQL (Parameter 3) should be zero. If it is not, the driver's cancellation routine may have caused this bug check by returning at an elevated IRQL. In this case, carefully note what was running (and what was closing) at the time of the crash, and note all of the installed drivers at the time of the crash. The cause in this case is usually a severe bug in a driver.

Resolution

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause.

Bug Check 0x21: QUOTA_UNDERFLOW

7/2/2019 • 2 minutes to read • Edit Online

The QUOTA_UNDERFLOW bug check has a value of 0x00000021. This indicates that quota charges have been mishandled by returning more quota to a particular block than was previously charged.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

QUOTA_UNDERFLOW Parameters

PARAMETER	DESCRIPTION
1	The process that was initially charged, if available.
2	The quota type. For the list of all possible quota type values, see the header file Ps.h in the Windows Driver Kit (WDK).
3	The initial charged amount of quota to return.
4	The remaining amount of quota that was not returned.

Bug Check 0x22: FILE_SYSTEM

7/2/2019 • 2 minutes to read • Edit Online

The FILE_SYSTEM bug check has a value of 0x00000022.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Resolution

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause.

Bug Check 0x23: FAT_FILE_SYSTEM

7/2/2019 • 2 minutes to read • Edit Online

The FAT_FILE_SYSTEM bug check has a value of 0x00000023. This indicates that a problem occurred in the FAT file system.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

FAT_FILE_SYSTEM Parameters

PARAMETER	DESCRIPTION
1	Specifies source file and line number information. The high 16 bits (the first four hexadecimal digits after the "0x") identify the source file by its identifier number. The low 16 bits identify the source line in the file where the bug check occurred.
2	If **FatExceptionFilter** is on the stack, this parameter specifies the address of the exception record.
3	If **FatExceptionFilter** is on the stack, this parameter specifies the address of the context record.
4	Reserved

Cause

One possible cause of this bug check is disk corruption. Corruption in the file system or bad blocks (sectors) on the disk can induce this error. Corrupted SCSI and IDE drivers can also adversely affect the system's ability to read and write to the disk, thus causing the error.

Another possible cause is depletion of nonpaged pool memory. If the nonpaged pool memory is completely depleted, this error can stop the system. However, during the indexing process, if the amount of available nonpaged pool memory is very low, another kernel-mode driver requiring nonpaged pool memory can also trigger this error.

Resolution

To debug this problem: Use the **.cxr (Display Context Record)** command with Parameter 3, and then use **kb (Display Stack Backtrace)**.

To resolve a disk corruption problem: Check Event Viewer for error messages from SCSI and FASTFAT (System Log) or Autochk (Application Log) that might help pinpoint the device or driver that is causing the error. Try disabling any virus scanners, backup programs, or disk defragmenter tools that continually monitor the system.

You should also run hardware diagnostics supplied by the system manufacturer. For details on these procedures, see the owner's manual for your computer. Run **Chkdsk /f /r** to detect and resolve any file system structural corruption. You must restart the system before the disk scan begins on a system partition.

To resolve a nonpaged pool memory depletion problem: Add new physical memory to the computer. This will increase the quantity of nonpaged pool memory available to the kernel.

Bug Check 0x24: NTFS_FILE_SYSTEM

7/2/2019 • 3 minutes to read • Edit Online

The NTFS_FILE_SYSTEM bug check has a value of 0x00000024. This indicates a problem occurred in ntfs.sys, the driver file that allows the system to read and write to NTFS drives.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

NTFS_FILE_SYSTEM Parameters

PARAMETER	DESCRIPTION
1	Specifies source file and line number information. The high 16 bits (the first four hexadecimal digits after the "0x") identify the source file by its identifier number. The low 16 bits identify the source line in the file where the bug check occurred.
2	If **NtfsExceptionFilter** is on the stack, this parameter specifies the address of the exception record.
3	If **NtfsExceptionFilter** is on the stack, this parameter specifies the address of the context record.
4	Reserved

Cause

One possible cause of this bug check is disk corruption. Corruption in the NTFS file system or bad blocks (sectors) on the hard disk can induce this error. Corrupted hard drive (SATA/IDE) drivers can also adversely affect the system's ability to read and write to disk, thus causing the error.

Resolution

To debug this problem: Use the **.cxr (Display Context Record)** command with Parameter 3, and then use **kb (Display Stack Backtrace)**.

To resolve a disk corruption problem:

- Check Event Viewer for error messages related to the hard drive appearing in the System Log that might help pinpoint the device or driver that is causing the error.
- Try disabling any virus scanners, backup programs, or disk defragmenter tools that continually monitor the system.

- You should also run hardware diagnostics supplied by the system manufacturer related to the storage sub system.
- Use the scan disk utility to confirm that there are no file system errors. Right click on the drive you want to scan and select **Properties**. Click on **Tools**. Click the **Check now** button.
- Confirm that there is sufficient free space on the hard drive. The operating system and some applications require sufficient free space to create swap files and for other functions. Based on the system configuration, the exact requirement varies, but it is normally a good idea to have 10% to 15% free space available.
- Use the System File Checker tool to repair missing or corrupted system files. The System File Checker is a utility in Windows that allows users to scan for corruptions in Windows system files and restore corrupted files. Use the following command to run the System File Checker tool (SFC.exe).

  ```
  SFC /scannow
  ```

  For more information, see Use the System File Checker tool to repair missing or corrupted system files.
- **Driver Verifier**

  Driver Verifier is a tool that runs in real time to examine the behavior of drivers. If it see errors in the execution of driver code, it proactively creates an exception to allow that part of the driver code to be further scrutinized. The driver verifier manager is built into Windows and is available on all Windows PCs. To start the driver verifier manager, type *Verifer* at a command prompt. You can configure which drivers you would like to verify. The code that verifies drivers adds overhead as it runs, so try and verify the smallest number of drivers as possible. For more information, see Driver Verifier.

In the past, another possible cause of this stop code is depletion of nonpaged pool memory. If the nonpaged pool memory is completely depleted, this error can stop the system. However, during the indexing process, if the amount of available nonpaged pool memory is very low, another kernel-mode driver requiring nonpaged pool memory can also trigger this error.

Bug Check 0x25: NPFS_FILE_SYSTEM

7/2/2019 • 2 minutes to read • Edit Online

The NPFS_FILE_SYSTEM bug check has a value of 0x00000025. This indicates that a problem occurred in the NPFS file system.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

NPFS_FILE_SYSTEM Parameters

PARAMETER	DESCRIPTION
1	Specifies source file and line number information. The high 16 bits (the first four hexadecimal digits after the "0x") identify the source file by its identifier number. The low 16 bits identify the source line in the file where the bug check occurred.
2	Reserved
3	Reserved
4	Reserved

Cause

One possible cause of this bug check is depletion of nonpaged pool memory. If the nonpaged pool memory is completely depleted, this error can stop the system. However, during the indexing process, if the amount of available nonpaged pool memory is very low, another kernel-mode driver requiring nonpaged pool memory can also trigger this error.

Resolution

To resolve a nonpaged pool memory depletion problem: Add new physical memory to the computer. This will increase the quantity of nonpaged pool memory available to the kernel.

Bug Check 0x26: CDFS_FILE_SYSTEM

7/2/2019 • 2 minutes to read • Edit Online

The CDFS_FILE_SYSTEM bug check has a value of 0x00000026. This indicates that a problem occurred in the CD file system.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

CDFS_FILE_SYSTEM Parameters

PARAMETER	DESCRIPTION
1	Specifies source file and line number information. The high 16 bits (the first four hexadecimal digits after the "0x") identify the source file by its identifier number. The low 16 bits identify the source line in the file where the bug check occurred.
2	If **CdExceptionFilter** is on the stack, this parameter specifies the address of the exception record.
3	If **CdExceptionFilter** is on the stack, this parameter specifies the address of the context record.
4	Reserved

Cause

One possible cause of this bug check is disk corruption. Corruption in the file system or bad blocks (sectors) on the disk can induce this error. Corrupted SCSI and IDE drivers can also adversely affect the system's ability to read and write to the disk, thus causing the error.

Another possible cause is depletion of nonpaged pool memory. If the nonpaged pool memory is completely depleted, this error can stop the system. However, during the indexing process, if the amount of available nonpaged pool memory is very low, another kernel-mode driver requiring nonpaged pool memory can also trigger this error.

Resolution

To debug this problem: Use the **.cxr (Display Context Record)** command with Parameter 3, and then use **kb (Display Stack Backtrace)**.

To resolve a disk corruption problem: Check Event Viewer for error messages from SCSI and FASTFAT (System Log) or Autochk (Application Log) that might help pinpoint the device or driver that is causing the error. Try disabling any virus scanners, backup programs, or disk defragmenter tools that continually monitor the system.

You should also run hardware diagnostics supplied by the system manufacturer. For details on these procedures, see the owner's manual for your computer. Run **Chkdsk /f /r** to detect and resolve any file system structural corruption. You must restart the system before the disk scan begins on a system partition.

To resolve a nonpaged pool memory depletion problem: Add new physical memory to the computer. This will increase the quantity of nonpaged pool memory available to the kernel.

Bug Check 0x27: RDR_FILE_SYSTEM

7/2/2019 • 2 minutes to read • Edit Online

The RDR_FILE_SYSTEM bug check has a value of 0x00000027. This indicates that a problem occurred in the SMB redirector file system.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

RDR_FILE_SYSTEM Parameters

PARAMETER	DESCRIPTION
1	The high 16 bits (the first four hexadecimal digits after the "0x") identify the type of problem. Possible values include: 0xCA550000 RDBSS_BUG_CHECK_CACHESUP 0xC1EE0000 RDBSS_BUG_CHECK_CLEANUP 0xC10E0000 RDBSS_BUG_CHECK_CLOSE 0xBAAD0000 RDBSS_BUG_CHECK_NTEXCEPT
2	If **RxExceptionFilter** is on the stack, this parameter specifies the address of the exception record.
3	If **RxExceptionFilter** is on the stack, this parameter specifies the address of the context record.
4	Reserved

Cause

One possible cause of this bug check is depletion of nonpaged pool memory. If the nonpaged pool memory is completely depleted, this error can stop the system. However, during the indexing process, if the amount of available nonpaged pool memory is very low, another kernel-mode driver requiring nonpaged pool memory can also trigger this error.

Resolution

To debug this problem: Use the **.cxr (Display Context Record)** command with Parameter 3, and then use **kb (Display Stack Backtrace)**.

To resolve a nonpaged pool memory depletion problem: Add new physical memory to the computer. This will increase the quantity of nonpaged pool memory available to the kernel.

Bug Check 0x28: CORRUPT_ACCESS_TOKEN

7/2/2019 • 2 minutes to read • Edit Online

The CORRUPT_ACCESS_TOKEN bug check has a value of 0x00000028.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Resolution

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause.

Bug Check 0x29: SECURITY_SYSTEM

7/2/2019 • 2 minutes to read • Edit Online

The SECURITY_SYSTEM bug check has a value of 0x00000029.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Resolution

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause.

Bug Check 0x2A: INCONSISTENT_IRP

7/2/2019 • 2 minutes to read • Edit Online

The INCONSISTENT_IRP bug check has a value of 0x0000002A. This indicates that an IRP was found to contain inconsistent information.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

INCONSISTENT_IRP Parameters

PARAMETER	DESCRIPTION
1	The address of the IRP that was found to be inconsistent
2	Reserved
3	Reserved
4	Reserved

Cause

An IRP was discovered to be in an inconsistent state. Usually this means some field of the IRP was inconsistent with the remaining state of the IRP. An example would be an IRP that was being completed, but was still marked as being queued to a driver's device queue.

Remarks

This bug check code is not currently being used in the system, but exists for debugging purposes.

Bug Check 0x2B: PANIC_STACK_SWITCH

7/2/2019 • 2 minutes to read • Edit Online

The PANIC_STACK_SWITCH bug check has a value of 0x0000002B. This indicates that the kernel mode stack was overrun.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

PANIC_STACK_SWITCH Parameters

PARAMETER	DESCRIPTION
1	The trap frame
2	Reserved
3	Reserved
4	Reserved

Cause

This error normally appears when a kernel-mode driver uses too much stack space. It can also appear when serious data corruption occurs in the kernel.

Resolution

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause.

Bug Check 0x2C: PORT_DRIVER_INTERNAL

7/2/2019 • 2 minutes to read • Edit Online

The PORT_DRIVER_INTERNAL bug check has a value of 0x0000002C.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Resolution

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause.

Bug Check 0x2D: SCSI_DISK_DRIVER_INTERNAL

7/2/2019 • 2 minutes to read • Edit Online

The SCSI_DISK_DRIVER_INTERNAL bug check has a value of 0x0000002D.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Resolution

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause.

Bug Check 0x2E: DATA_BUS_ERROR

7/2/2019 • 2 minutes to read • Edit Online

The DATA_BUS_ERROR bug check has a value of 0x0000002E. This typically indicates that a parity error in system memory has been detected.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

DATA_BUS_ERROR Parameters

PARAMETER	DESCRIPTION
1	Virtual address that caused the fault
2	Physical address that caused the fault
3	Processor status register (PSR)
4	Faulting instruction register (FIR)

Cause

This error is almost always caused by a hardware problem -- a configuration issue, defective hardware, or incompatible hardware.

The most common hardware problems that can cause this error are defective RAM, Level 2 (L2) RAM cache errors, or video RAM errors. Hard disk corruption can also cause this error.

This bug check can also be caused when a device driver attempts to access an address in the 0x8*xxxxxxx* range that does not exist (in other words, that does not have a physical address mapping).

Resolution

Resolving a hardware problem: If hardware has recently been added to the system, remove it to see if the error recurs.

If existing hardware has failed, remove or replace the faulty component. You should run hardware diagnostics supplied by the system manufacturer to determine which hardware component has failed. For details on these procedures, see the owner's manual for your computer. Check that all adapter cards in the computer are properly seated. Use an ink eraser or an electrical contact treatment, available at electronics supply stores, to ensure that adapter card contacts are clean.

If the problem occurs on a newly installed system, check the availability of updates for the BIOS, the SCSI controller or network cards. Updates of this kind are typically available on the Web site or the bulletin board

system (BBS) of the hardware manufacturer.

If the error occurs after installing a new or updated device driver, the driver should be removed or replaced. If, under this circumstance, the error occurs during startup and the system partition is formatted with NTFS, you might be able to use Safe Mode to rename or delete the faulty driver.

If the driver is used as part of the system startup process in Safe Mode, you need to start the computer using the Recovery Console in order to access the file.

For additional error messages that might help pinpoint the device or driver that is causing the error, check the System Log in Event Viewer. Disabling memory caching or shadowing in the BIOS might also resolve this error. In addition, check the system for viruses, using any up-to-date commercial virus scanning software that examines the Master Boot Record of the hard disk. All Windows file systems can be infected by viruses.

Resolving a hard disk corruption problem: Run **Chkdsk /f /r** on the system partition. You must restart the system before the disk scan begins. If you cannot start the system due to the error, use the Recovery Console and run **Chkdsk /r**.

Warning If your system partition is formatted with the file allocation table (FAT) file system, the long filenames used by Windows can be damaged if Scandisk or another Microsoft MS-DOS-based hard disk tool is used to verify the integrity of your hard disk from MS-DOS. Always use the version of Chkdsk that matches your Windows version.

Bug Check 0x2F: INSTRUCTION_BUS_ERROR

7/2/2019 • 2 minutes to read • Edit Online

The INSTRUCTION_BUS_ERROR bug check has a value of 0x0000002F.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Resolution

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause.

Bug Check 0x30: SET_OF_INVALID_CONTEXT

7/2/2019 • 2 minutes to read • Edit Online

The SET_OF_INVALID_CONTEXT bug check has a value of 0x00000030. This indicates that the stack pointer in a trap frame had an invalid value.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

SET_OF_INVALID_CONTEXT Parameters

PARAMETER	DESCRIPTION
1	The new stack pointer
2	The old stack pointer
3	The trap frame address
4	0

Cause

This bug check occurs when some routine attempts to set the stack pointer in the trap frame to a lower value than the current stack pointer value.

If this error were not caught, it would cause the kernel to run with a stack pointer pointing to stack which is no longer valid.

Bug Check 0x31: PHASE0_INITIALIZATION_FAILED

7/2/2019 • 2 minutes to read • Edit Online

The PHASE0_INITIALIZATION_FAILED bug check has a value of 0x00000031. This indicates that system initialization failed.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

PHASE0_INITIALIZATION_FAILED Parameters

None

Cause

System initialization failed at a very early stage.

Resolution

A debugger is required to analyze this.

Bug Check 0x32: PHASE1_INITIALIZATION_FAILED

7/2/2019 • 2 minutes to read • Edit Online

The PHASE1_INITIALIZATION_FAILED bug check has a value of 0x00000032. This indicates that system initialization failed.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

PHASE1_INITIALIZATION_FAILED Parameters

PARAMETER	DESCRIPTION
1	The NT status code that describes why the system initialization failed
2	Reserved
3	Reserved
4	Reserved

Bug Check 0x33: UNEXPECTED_INITIALIZATION_CALL

7/2/2019 • 2 minutes to read • Edit Online

The UNEXPECTED_INITIALIZATION_CALL bug check has a value of 0x00000033.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Bug Check 0x34: CACHE_MANAGER

7/2/2019 • 2 minutes to read • Edit Online

The CACHE_MANAGER bug check has a value of 0x00000034. This indicates that a problem occurred in the file system's cache manager.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

CACHE_MANAGER Parameters

PARAMETER	DESCRIPTION
1	Specifies source file and line number information. The high 16 bits (the first four hexadecimal digits after the "0x") identify the source file by its identifier number. The low 16 bits identify the source line in the file where the bug check occurred.
2	Reserved
3	Reserved
4	Reserved

Cause

One possible cause of this bug check is depletion of nonpaged pool memory. If the nonpaged pool memory is completely depleted, this error can stop the system. However, during the indexing process, if the amount of available nonpaged pool memory is very low, another kernel-mode driver requiring nonpaged pool memory can also trigger this error.

Resolution

To resolve a nonpaged pool memory depletion problem: Add new physical memory to the computer. This will increase the quantity of nonpaged pool memory available to the kernel.

Bug Check 0x35: NO_MORE_IRP_STACK_LOCATIONS

7/2/2019 • 2 minutes to read • Edit Online

The NO_MORE_IRP_STACK_LOCATIONS bug check has a value of 0x00000035. This bug check occurs when the **IoCallDriver** packet has no more stack locations remaining.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

NO_MORE_IRP_STACK_LOCATIONS Parameters

PARAMETER	DESCRIPTION
1	Address of the IRP
2	Reserved
3	Reserved
4	Reserved

Cause

A higher-level driver has attempted to call a lower-level driver through the **IoCallDriver** interface, but there are no more stack locations in the packet. This will prevent the lower-level driver from accessing its parameters.

This is a disastrous situation, since the higher level driver is proceeding as if it has filled in the parameters for the lower level driver (as required). But since there is no stack location for the latter driver, the former has actually written off the end of the packet. This means that some other memory has been corrupted as well.

Bug Check 0x36: DEVICE_REFERENCE_COUNT_NOT_ZERO

7/2/2019 • 2 minutes to read • Edit Online

The DEVICE_REFERENCE_COUNT_NOT_ZERO bug check has a value of 0x00000036. This indicates that a driver attempted to delete a device object that still had a positive reference count.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

DEVICE_REFERENCE_COUNT_NOT_ZERO Parameters

PARAMETER	DESCRIPTION
1	The address of the device object
2	Reserved
3	Reserved
4	Reserved

Cause

A device driver has attempted to delete one of its device objects from the system, but the reference count for that object was non-zero.

This means there are still outstanding references to the device. (The reference count indicates the number of reasons why this device object cannot be deleted.)

This is a bug in the calling device driver.

Bug Check 0x37: FLOPPY_INTERNAL_ERROR

7/2/2019 • 2 minutes to read • Edit Online

The FLOPPY_INTERNAL_ERROR bug check has a value of 0x00000037.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Bug Check 0x38: SERIAL_DRIVER_INTERNAL

7/2/2019 • 2 minutes to read • Edit Online

The SERIAL_DRIVER_INTERNAL bug check has a value of 0x00000038.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Bug Check 0x39: SYSTEM_EXIT_OWNED_MUTEX

7/2/2019 • 2 minutes to read • Edit Online

The SYSTEM_EXIT_OWNED_MUTEX bug check has a value of 0x00000039. This indicates that the worker routine returned without releasing the mutex object that it owned.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

SYSTEM_EXIT_OWNED_MUTEX Parameters

PARAMETER	DESCRIPTION
1	The address of the worker routine that caused the error.
2	The parameter passed to the worker routine.
3	The address of the work item.
4	Reserved.

Cause

The worker routine returned while it still owned a mutex object. The current worker thread will proceed to run other unrelated work items, and the mutex will never be released.

Resolution

A debugger is required to analyze this problem. To find the driver that caused the error, use the **ln** (List Nearest Symbols) debugger command:

```
kd> ln address
```

Where address is the worker routine given in Parameter 1.

Bug Check 0x3A: SYSTEM_UNWIND_PREVIOUS_USER

7/2/2019 • 2 minutes to read • Edit Online

The SYSTEM_UNWIND_PREVIOUS_USER bug check has a value of 0x0000003A.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Bug Check 0x3B: SYSTEM_SERVICE_EXCEPTION

10/15/2019 • 4 minutes to read • Edit Online

The SYSTEM_SERVICE_EXCEPTION bug check has a value of 0x0000003B. This indicates that an exception happened while executing a routine that transitions from non-privileged code to privileged code.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

SYSTEM_SERVICE_EXCEPTION parameters

PARAMETER	DESCRIPTION
1	The exception that caused the bug check.
2	The address of the instruction that caused the bug check
3	The address of the context record for the exception that caused the bug check
4	0

Cause

This stop code indicates that executing code had an exception, and the thread that was below it is a system thread.

The exception information that is returned in parameter 1 is described in NTSTATUS values. The exception codes are defined in *ntstatus.h*, a header file provided by the Windows Driver Kit. (For more info, see Header files in the Windows Driver Kit).

Common exception codes include:

- 0x80000003: STATUS_BREAKPOINT

  A breakpoint or ASSERT was encountered when no kernel debugger was attached to the system.

- 0xC0000005: STATUS_ACCESS_VIOLATION

  A memory access violation occurred. (Parameter 4 of the bug check is the address that the driver attempted to access.)

Resolution

To debug this problem, use the **.cxr** (display context record) command with Parameter 3, and then use **kb** (display stack backtrace). You can also set a breakpoint in the code that precedes this stop code and attempt to single-step forward into the faulting code. Use the **u**, **ub**, **uu** (unassemble) commands to see the assembly program code.

The **!analyze** debugger extension displays information about the bug check and can be helpful in determining the root cause. The following example is output from **!analyze**.

```
SYSTEM_SERVICE_EXCEPTION (3b)
An exception happened while executing a system service routine.
Arguments:
Arg1: 00000000c0000005, Exception code that caused the bugcheck
Arg2: fffff802328375b0, Address of the instruction which caused the bugcheck
Arg3: ffff9c0a746c2330, Address of the context record for the exception that caused the bugcheck
Arg4: 0000000000000000, zero.
...
```

For more information about WinDbg and **!analyze**, see the following topics:

- Using the !analyze extension
- Analyzing a kernel-mode dump file with WinDbg

Identify the driver

If a driver that is responsible for the error can be identified, its name is printed on the blue screen and stored in memory at the location (PUNICODE_STRING) **KiBugCheckDriver**. You can use **dx** (display debugger object model expression), a debugger command, to display this: `dx KiBugCheckDriver`.

Use the **!error** extension to display information about the exception code in parameter 1. Following is an example of output from **!error**.

```
2: kd> !error 00000000c0000005
Error code: (NTSTATUS) 0xc0000005 (3221225477) - The instruction at 0x%p referenced memory at 0x%p. The memory
could not be %s.
```

Look at the **STACK TEXT** output from WinDbg for clues about what was running when the failure occurred. If multiple dump files are available, compare their information to look for common code that is in the stack. Use debugger commands like **kb** (display stack backtrace) to investigate the faulting code.

Use the following command to list modules that are loaded in memory: **lm t n**

Use **!memusage** to examine the general state of the system memory. You can also use the commands **!pte** and **!pool** to examine specific areas of memory.

In the past, this error has been linked to excessive use of the paged pool, which may occur due to user-mode graphics drivers crossing over and passing bad data to the kernel code. If you suspect this is the case, use the pool options in Driver Verifier to gather additional information.

Driver Verifier

Driver Verifier is a tool that runs in real time to examine the behavior of drivers. For example, Driver Verifier checks the use of memory resources, such as memory pools. If it identifies errors in the execution of driver code, it proactively creates an exception to allow that part of the driver code to be further scrutinized. Driver Verifier Manager is built into Windows and is available on all Windows PCs.

To start Driver Verifier Manager, enter **verifier** at a command prompt. You can configure which drivers to verify. The code that verifies drivers adds overhead as it runs, so try to verify the smallest number of drivers possible. For more information, see Driver Verifier.

Remarks

For general troubleshooting of Windows bug check codes, follow these suggestions:

- If new device drivers or system services have been added recently, try removing or updating them. Try to determine what changed in the system that caused the new bug check code to appear.
- Look in Device Manager to see if any devices are marked with an exclamation point (!), which indicates a problem. Review the events log displayed in the properties for any faulting device driver. Try to update the related driver.
- Check the System Log in Event Viewer for additional error messages that might help pinpoint the device or driver that is causing the error. Look for critical errors in the system log that occurred in the same time window as the blue screen.
- If you recently added hardware to the system, try removing or replacing it. Or check with the manufacturer to see if any patches are available.

For additional general troubleshooting information, see Blue screen data.

Bug Check 0x3C: INTERRUPT_UNWIND_ATTEMPTED

7/2/2019 • 2 minutes to read • Edit Online

The INTERRUPT_UNWIND_ATTEMPTED bug check has a value of 0x0000003C.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Bug Check 0x3D: INTERRUPT_EXCEPTION_NOT_HANDLED

11/22/2019 • 2 minutes to read • Edit Online

The INTERRUPT_EXCEPTION_NOT_HANDLED bug check has a value of 0x0000003D.

This bug check appears very infrequently.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Resolution

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause.

Bug Check 0x3E: MULTIPROCESSOR_CONFIGURATION_NOT_SUPPORTED

7/2/2019 • 2 minutes to read • Edit Online

The MULTIPROCESSOR_CONFIGURATION_NOT_SUPPORTED bug check has a value of 0x0000003E. This indicates that the system has multiple processors, but they are asymmetric in relation to one another.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

MULTIPROCESSOR_CONFIGURATION_NOT_SUPPORTED Parameters

None

Cause

In order to be symmetric, all processors must be of the same type and level. This system contains processors of different types (for example, a Pentium processor and an 80486 processor).

Bug Check 0x3F: NO_MORE_SYSTEM_PTES

7/2/2019 • 2 minutes to read • Edit Online

The NO_MORE_SYSTEM_PTES bug check has a value of 0x0000003F. This is the result of a system which has performed too many I/O actions. This has resulted in fragmented system page table entries (PTE).

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

NO_MORE_SYSTEM_PTES Parameters

PARAMETER	DESCRIPTION
1	**0:** system expansion PTE type **1:** nonpaged pool expansion PTE type
2	Size of memory request
3	Total free system PTEs
4	Total system PTEs

Cause

In almost all cases, the system is not actually out of PTEs. Rather, a driver has requested a large block of memory, but there is no contiguous block of sufficient size to satisfy this request.

Often video drivers will allocate large amounts of kernel memory that must succeed. Some backup programs do the same.

Resolution

A possible work-around: Modify the registry to increase the total number of system PTEs. If this does not help, remove any recently-installed software, especially backup utilities or disk-intensive applications.

Debugging the problem: The following method can be used to debug bug check 0x3F.

First, get a stack trace, and use the **!sysptes 3** extension command.

Then set **HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\TrackPtes** equal to DWORD 1, and reboot. This will cause the system to save stack traces.

This allows you to display more detailed information about the PTE owners. For example:

```
0: kd> !sysptes 4

0x2c47 System PTEs allocated to mapping locked pages

VA       MDL     PageCount  Caller/CallersCaller
f0e5db48 eb6ceef0        1 ntkrpamp!MmMapLockedPages+0x15/ntkrpamp!IopfCallDriver+0x35
f0c3fe48 eb634bf0        1 netbt!NbtTdiAssociateConnection+0x1f/netbt!DelayedNbtProcessConnect+0x17c
f0db38e8 eb65b880        1 mrxsmb!SmbMmAllocateSessionEntry+0x89/mrxsmb!SmbCepInitializeExchange+0xda
f8312568 eb6df880        1 rdbss!RxCreateFromNetRoot+0x3d7/rdbss!RxCreateFromNetRoot+0x93
f8363908 eb685880        1 mrxsmb!SmbMmAllocateSessionEntry+0x89/mrxsmb!SmbCepInitializeExchange+0xda
f0c54248 eb640880        1 rdbss!RxCreateFromNetRoot+0x3d7/rdbss!RxCreateFromNetRoot+0x93
f0ddf448 eb5f3160        1 mrxsmb!MrxSmbUnalignedDirEntryCopyTail+0x387/mrxsmb!MRxSmbCoreInformation+0x36
f150bc08 eb6367b0        1 mrxsmb!MrxSmbUnalignedDirEntryCopyTail+0x387/mrxsmb!MRxSmbCoreInformation+0x36
f1392308 eb6fba70        1 netbt!NbtTdiOpenAddress+0x1fb/netbt!DelayedNbtProcessConnect+0x17c
eb1bee64 edac5000      200 VIDEOPRT!pVideoPortGetDeviceBase+0x118/VIDEOPRT!VideoPortMapMemory+0x45
f139b5a8 edd4b000       12 rdbss!FsRtlCopyWrite2+0x34/rdbss!RxDriverEntry+0x149
eb41f400 ede92000       20 VIDEOPRT!pVideoPortGetDeviceBase+0x139/VIDEOPRT!VideoPortGetDeviceBase+0x1b
eb41f198 edf2a000       20 NDIS!NdisReadNetworkAddress+0x3a/NDIS!NdisFreeSharedMemory+0x58
eb41f1e4 eb110000       10 VIDEOPRT!pVideoPortGetDeviceBase+0x139/VIDEOPRT!VideoPortGetDeviceBase+0x1b
......
```

If the system runs out of PTEs again after the **TrackPtes** registry value has been set, **bug check 0xD8** (DRIVER_USED_EXCESSIVE_PTES) will be issued instead of 0x3F. The name of the driver causing this error will be displayed as well.

Bug Check 0x40: TARGET_MDL_TOO_SMALL

7/2/2019 • 2 minutes to read • Edit Online

The TARGET_MDL_TOO_SMALL bug check has a value of 0x00000040. This indicates that a driver has improperly used **IoBuildPartialMdl**.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

TARGET_MDL_TOO_SMALL Parameters

None

Cause

This is a driver bug. A driver has called the **IoBuildPartialMdl** function and passed it an MDL to map part of a source MDL, but the target MDL is not large enough to map the entire range of addresses requested.

Resolution

The source and target MDLs, as well as the address range length to be mapped, are the first, second, and fourth arguments to the **IoBuildPartialMdl** function. Therefore, doing a stack trace on this particular function might help during the debugging process. Ensure that your code is correctly calculating the necessary size for the target MDL for the address range length that you are passing to this function.

Bug Check 0x41: MUST_SUCCEED_POOL_EMPTY

7/2/2019 • 2 minutes to read • Edit Online

The MUST_SUCCEED_POOL_EMPTY bug check has a value of 0x00000041. This indicates that a kernel-mode thread has requested too much must-succeed pool.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

MUST_SUCCEED_POOL_EMPTY Parameters

PARAMETER	DESCRIPTION
1	The size of the request that could not be satisfied
2	The number of pages used from nonpaged pool
3	The number of requests from nonpaged pool larger than PAGE_SIZE
4	The number of pages available

Cause

No driver is permitted to request must-succeed pool.

If a must-succeed request cannot be filled, this bug check is issued.

Resolution

Replace or rewrite the driver which is making the request. A driver should not request must-succeed pool. Instead, it should ask for normal pool and gracefully handle the scenario where the pool is temporarily empty.

The **kb (Display Stack Backtrace)** command will show the driver that caused the error.

Additionally, it is possible that a second component has depleted the must-succeed pool. To determine if this is the case, first use the **kb** command. Then use **!vm 1** to display total pool usage, **!poolused 2** to display per-tag nonpaged pool usage, and **!poolused 4** to display per-tag paged pool usage. The component associated with the tag using the most pool is probably the source of the problem.

Bug Check 0x42: ATDISK_DRIVER_INTERNAL

7/2/2019 • 2 minutes to read • Edit Online

The ATDISK_DRIVER_INTERNAL bug check has a value of 0x00000042.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Bug Check 0x43: NO_SUCH_PARTITION

7/2/2019 • 2 minutes to read • Edit Online

The NO_SUCH_PARTITION bug check has a value of 0x00000043.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Bug Check 0x44: MULTIPLE_IRP_COMPLETE_REQUESTS

7/2/2019 • 2 minutes to read • Edit Online

The MULTIPLE_IRP_COMPLETE_REQUESTS bug check has a value of 0x00000044. This indicates that a driver has tried to request an IRP be completed that is already complete.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

MULTIPLE_IRP_COMPLETE_REQUESTS Parameters

PARAMETER	DESCRIPTION
1	The address of the IRP
2	Reserved
3	Reserved
4	Reserved

Cause

A driver has called **IoCompleteRequest** to ask that an IRP be completed, but the packet has already been completed.

Resolution

This is a tough bug to find because the simplest case -- a driver that attempted to complete its own packet twice -- is usually not the source of the problem. More likely, two separate drivers each believe that they own the packet, and each has attempted to complete it. The first request succeeds, and the second fails, resulting in this bug check.

Tracking down which drivers in the system caused the error is difficult, because the trail of the first driver has been covered by the second. However, the driver stack for the current request can be found by examining the device object fields in each of the stack locations.

Bug Check 0x45: INSUFFICIENT_SYSTEM_MAP_REGS

7/2/2019 • 2 minutes to read • Edit Online

The INSUFFICIENT_SYSTEM_MAP_REGS bug check has a value of 0x00000045.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Bug Check 0x46: DEREF_UNKNOWN_LOGON_SESSION

7/2/2019 • 2 minutes to read • Edit Online

The DEREF_UNKNOWN_LOGON_SESSION bug check has a value of 0x00000046.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Bug Check 0x47: REF_UNKNOWN_LOGON_SESSION

7/2/2019 • 2 minutes to read • Edit Online

The REF_UNKNOWN_LOGON_SESSION bug check has a value of 0x00000047.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Bug Check 0x48: CANCEL_STATE_IN_COMPLETED_IRP

7/2/2019 • 2 minutes to read • Edit Online

The CANCEL_STATE_IN_COMPLETED_IRP bug check has a value of 0x00000048. This indicates that an I/O request packet (IRP) was completed, and then was subsequently canceled.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

CANCEL_STATE_IN_COMPLETED_IRP Parameters

PARAMETER	DESCRIPTION
1	A pointer to the IRP
2	The cancel routine set by the driver
3	Reserved
4	Reserved

Cause

An IRP that had a *Cancel* routine set was completed normally, without cancellation. But after it was complete, a driver called the IRP's *Cancel* routine.

This could be caused by a driver that completed the IRP and then attempted to cancel it.

It could also be caused by two drivers each trying to access the same IRP in an improper way.

Resolution

The cancel routine parameter can be used to determine which driver or stack caused the bug check.

Bug Check 0x49: PAGE_FAULT_WITH_INTERRUPTS_OFF

11/22/2019 • 2 minutes to read • Edit Online

The PAGE_FAULT_WITH_INTERRUPTS_OFF bug check has a value of 0x00000049.

This bug check appears very infrequently.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Resolution

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause.

Bug Check 0x4A: IRQL_GT_ZERO_AT_SYSTEM_SERVICE

7/2/2019 • 2 minutes to read • Edit Online

The IRQL_GT_ZERO_AT_SYSTEM_SERVICE bug check has a value of 0x0000004A. This indicates that a thread is returning to user mode from a system call when its IRQL is still above PASSIVE_LEVEL.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

IRQL_GT_ZERO_AT_SYSTEM_SERVICE Parameters

PARAMETER	DESCRIPTION
1	The address of the system function (system call routine)
2	The current IRQL
3	0
4	0

Resolution

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause.

Bug Check 0x4B: STREAMS_INTERNAL_ERROR

7/2/2019 • 2 minutes to read • Edit Online

The STREAMS_INTERNAL_ERROR bug check has a value of 0x0000004B.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Bug Check 0x4C: FATAL_UNHANDLED_HARD_ERROR

7/2/2019 • 2 minutes to read • Edit Online

The FATAL_UNHANDLED_HARD_ERROR bug check has a value of 0x0000004C.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Bug Check 0x4D: NO_PAGES_AVAILABLE

7/2/2019 • 2 minutes to read • Edit Online

The NO_PAGES_AVAILABLE bug check has a value of 0x0000004D. This indicates that no free pages are available to continue operations.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

NO_PAGES_AVAILABLE Parameters

PARAMETER	DESCRIPTION
1	The total number of dirty pages
2	The number of dirty pages destined for the page file
3	The size of the nonpaged pool available at the time the bug check occurred
4	The most recent modified write error status.

Cause

To see general memory statistics, use the **!vm 3** extension.

This bug check can occur for any of the following reasons:

- A driver has blocked, deadlocking the modified or mapped page writers. Examples of this include mutex deadlocks or accesses to paged out memory in file system drivers or filter drivers. This indicates a driver bug.

  If Parameter 1 or Parameter 2 is large, then this is a possibility. Use **!vm 3**.

- A storage driver is not processing requests. Examples of this are stranded queues and non-responding drives. This indicates a driver bug.

  If Parameter 1 or Parameter 2 is large, then this is a possibility. Use **!vm 8**, followed by **!process 0 7**.

- A high-priority realtime thread has starved the balance set manager from trimming pages from the working set, or starved the modified page writer from writing them out. This indicates a bug in the component that created this thread.

  This situation is difficult to analyze. Try using **!ready**. Try also **!process 0 7** to list all threads and see if any have accumulated excessive kernel time as well as what their current priorities are. Such processes may have blocked out the memory management threads from making pages available.

- Not enough pool is available for the storage stack to write out modified pages. This indicates a driver bug.

  If Parameter 3 is small, then this is a possibility. Use **!vm** and **!poolused 2**.

If the problem cannot be found, then try booting with a kernel debugger attached from the beginning, and monitor the situation.

Bug Check 0x4E: PFN_LIST_CORRUPT

7/2/2019 • 2 minutes to read • Edit Online

The PFN_LIST_CORRUPT bug check has a value of 0x0000004E. This indicates that the page frame number (PFN) list is corrupted.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

PFN_LIST_CORRUPT Parameters

Parameter 1 indicates the type of violation. The meaning of the other parameters depends on the value of *Parameter 1*.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x01	The **ListHead** value that was corrupted	The number of pages available	0	The list head was corrupted.
0x02	The entry in the list that is being removed	The highest physical page number	The reference count of the entry being removed	A list entry was corrupted.
0x07	The page frame number	The current share count	0	A driver has unlocked a certain page more times than it locked it.
0x8D	The page frame number whose state is inconsistent	0	0	The page-free list is corrupted. This error code most likely indicates a hardware issue.
0x8F	New page number	Old page number	0	The free or zeroed page listhead is corrupted.
0x99	Page frame number	Current page state	0	A page table entry (PTE) or PFN is corrupted.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x9A	Page frame number	Current page state	The reference count of the entry that is being removed	A driver attempted to free a page that is still locked for IO.

Cause

This error is typically caused by a driver passing a bad memory descriptor list. For example, the driver might have called **MmUnlockPages** twice with the same list.

If a kernel debugger is available, examine the stack trace: the **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause, then enter one of the **k (Display Stack Backtrace)** commands to view the call stack.

Bug Check 0x4F: NDIS_INTERNAL_ERROR

7/2/2019 • 2 minutes to read • Edit Online

The NDIS_INTERNAL_ERROR bug check has a value of 0x0000004F.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Bug Check 0x50: PAGE_FAULT_IN_NONPAGED_AREA

8/26/2019 • 6 minutes to read • Edit Online

The PAGE_FAULT_IN_NONPAGED_AREA bug check has a value of 0x00000050. This indicates that invalid system memory has been referenced. Typically the memory address is wrong or the memory address is pointing at freed memory.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

PAGE_FAULT_IN_NONPAGED_AREA Parameters

PARAMETER	DESCRIPTION
1	Memory address referenced
2	*After Windows 1507 (TH1) Version - x64* **0:** Read operation **2:** Write operation **10:** Execute operation *After Windows 1507 (TH1) Version - x86* **0:** Read operation **2:** Write operation **10:** Execute operation *After Windows 1507 (TH1) Version - ARM* **0:** Read operation **1:** Write operation **8:** Execute operation *Before Windows 1507 (TH1) Version x64 / x86* **0:** Read operation **1:** Write operation
3	Address that referenced memory (if known)

PARAMETER	DESCRIPTION
4	Type of page fault 0x03 - NONPAGED_BUGCHECK_WRONG_SESSION - An attempted reference to a session space address was made in the context of a process that has no session. Typically this means the caller is improperly trying to access a session address without correctly obtaining an object reference to the correct process and attaching to it first. This bugcheck & subtype was last used in Windows 10 RS3. In Windows 10 RS4 and above, this error is instead surfaced as 0x02 (NONPAGED_BUGCHECK_NOT_PRESENT_PAGE_TABLE). 0x04 - NONPAGED_BUGCHECK_VA_NOT_CANONICAL - An attempted reference to a non-canonical (illegal) virtual address (Parameter 1) was attempted. The caller should not ever be trying to access this address.

If the driver responsible for the error can be identified, its name is printed on the blue screen and stored in memory at the location (PUNICODE_STRING) **KiBugCheckDriver**. You can use the debugger dx command to display this - `dx KiBugCheckDriver`.

Cause

Bug check 0x50 can be caused by the installation of a faulty system service or faulty driver code. Antivirus software can also trigger this error, as can a corrupted NTFS volume.

It could also occur after the installation of faulty hardware or in the event of failure of installed hardware (usually related to defective RAM, be it main memory, L2 RAM cache, or video RAM).

Remarks

Event Log: Check the System Log in Event Viewer for additional error messages that might help pinpoint the device or driver that is causing the error. For more information, see Open Event Viewer. Look for critical errors in the system log that occurred in the same time window as the blue screen.

Resolving a faulty driver: Examine the name of the driver if that was listed on the blue screen or is present in the event log. Contact the driver vendor to see if an updated driver is available.

Resolving a faulty system service problem: Disable the service and confirm that this resolves the error. If so, contact the manufacturer of the system service about a possible update. If the error occurs during system startup, investigate the Windows repair options. For more information, see Recovery options in Windows 10.

Resolving an antivirus software problem: Disable the program and confirm that this resolves the error. If it does, contact the manufacturer of the program about a possible update.

Resolving a corrupted NTFS volume problem: Run **Chkdsk /f /r** to detect and repair disk errors. You must restart the system before the disk scan begins on a system partition. Contact the manufacture of the hard driver system to locate any diagnostic tools that they provide for the hard drive sub system.

Windows Memory Diagnostics: Run the Windows Memory Diagnostics tool, to test the physical memory. Click the Start button, and then clicking Control Panel. In the search box, type Memory, and then click *Diagnose your computer's memory problems*. After the test is run, use Event viewer to view the results under the System log. Look for the *MemoryDiagnostics-Results* entry to view the results.

Resolving a faulty hardware problem: If hardware has been added to the system recently, remove it to see if

the error recurs. If existing hardware has failed, remove or replace the faulty component. You should run hardware diagnostics supplied by the system manufacturer. For details on these procedures, see the owner's manual for your computer.

For general blue screen troubleshooting information, see **Blue Screen Data**.

Resolution

Typically, the referenced address is in freed memory or is simply invalid. This cannot be protected by a **try - except** handler -- it can only be protected by a probe or similar programming techniques.

Use the **!analyze** debug extension with the -v verbose option to display information about the bug check to work to determine the root cause.

```
2: kd> !analyze -v
*******************************************************************************
*                                                                             *
*                        Bugcheck Analysis                                    *
*                                                                             *
*******************************************************************************

PAGE_FAULT_IN_NONPAGED_AREA (50)
Invalid system memory was referenced.  This cannot be protected by try-except.
Typically the address is just plain bad or it is pointing at freed memory.
Arguments:
Arg1: ffffffff00000090, memory referenced.
Arg2: 0000000000000000, value 0 = read operation, 1 = write operation.
Arg3: fffff80240d322f9, If non-zero, the instruction address which referenced the bad memory
 address.
Arg4: 000000000000000c, (reserved)
```

In this example Parameter 2 indicates that the bug check occurred when an area of memory was being read.

Look at all of the !analyze output to gain information about what was going on when the bug check occurred. Examine MODULE_NAME: and the FAULTING_MODULE: to see which code is involved in referencing the invalid system memory.

Look at the STACK TEXT for clues on what was running when the failure occurred. If multiple dump files are available, compare information to look for common code that is in the stack.

Use the .trap command provided in the !analyze output to set the context.

```
TRAP_FRAME:  fffff98112e8b3d0 -- (.trap 0xfffff98112e8b3d0)
```

Use debugger commands such as use **kb (Display Stack Backtrace)** to investigate the faulting code.

Use the `lm t n` to list modules that are loaded in the memory.

Use the d, da, db, dc, dd, dD, df, dp, dq, du, dw (Display Memory) command to investigate the areas of memory referenced by parameter 1 and parameter 3.

```
2: kd> db ffffffff00000090
ffffffff`00000090  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
ffffffff`000000a0  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
ffffffff`000000b0  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
ffffffff`000000c0  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
ffffffff`000000d0  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
ffffffff`000000e0  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
ffffffff`000000f0  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
ffffffff`00000100  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
```

In this case doesn't look like there is data in this area of memory in parameter 1, which is the area of memory that was attempting to be read.

Use the !address command to look at parameter 3 which is the address of the the instruction which referenced the bad memory.

```
2: kd> !address fffff80240d322f9
Usage:                  Module
Base Address:           fffff802`40ca8000
End Address:            fffff802`415fb000
Region Size:            00000000`00953000
VA Type:                BootLoaded
Module name:            ntoskrnl.exe
Module path:            [\SystemRoot\system32\ntoskrnl.exe]
```

Use u, ub, uu (Unassemble)Dissasemble with parameter 3, to examine the which referenced the bad memory. For more information on X64 processor and assembly language see The x64 Processor.

```
2: kd> u fffff80240d322f9
nt!RtlSubtreePredecessor+0x9:
fffff802`40d322f9 488b4810        mov     rcx,qword ptr [rax+10h]
fffff802`40d322fd eb07            jmp     nt!RtlSubtreePredecessor+0x16 (fffff802`40d32306)
fffff802`40d322ff 488bc1          mov     rax,rcx
fffff802`40d32302 488b4910        mov     rcx,qword ptr [rcx+10h]
fffff802`40d32306 4885c9          test    rcx,rcx
fffff802`40d32309 75f4            jne     nt!RtlSubtreePredecessor+0xf (fffff802`40d322ff)
fffff802`40d3230b c3              ret
fffff802`40d3230c c3              ret
```

Use `ub` to dissassemble backwards from the a given address.

Use the r (Registers) command to examine what was being executed as the system bug checked.

```
2: kd> r
Last set context:
rax=ffffffff00000080 rbx=0000000000000000 rcx=ffffa68337cb7028
rdx=7a107838c48dfc00 rsi=0000000000000000 rdi=0000000000000000
rip=fffff80240d322f9 rsp=ffff840c96510958 rbp=ffffffffffffffff
 r8=ffffffffffffffff  r9=0000000000000000 r10=7ffffffffffffffc
r11=ffff840c96510a10 r12=0000000000000000 r13=0000000000000000
r14=0000000000000000 r15=0000000000000000
iopl=0         nv up ei ng nz na pe nc
cs=0010  ss=0018  ds=0000  es=0000  fs=0000  gs=0000             efl=00010282
nt!RtlSubtreePredecessor+0x9:
fffff802`40d322f9 488b4810        mov     rcx,qword ptr [rax+10h] ds:ffffffff`00000090=????????????????
```

In this case `fffff80240d322f9` is in the instruction pointer register, rip.

The `!pte` and `!pool` command may also be used to examine memory.

Use `!memusage` and to examine the general state of the system memory.

Driver Verifier

Driver Verifier is a tool that runs in real time to examine the behavior of drivers. For example, Driver Verifier checks the use of memory resources, such as memory pools. If it sees errors in the execution of driver code, it proactively creates an exception to allow that part of the driver code to be further scrutinized. The driver verifier manager is built into Windows and is available on all Windows PCs. To start the driver verifier manager, type *Verifer* at a command prompt. You can configure which drivers you would like to verify. The code that verifies drivers adds overhead as it runs, so try and verify the smallest number of drivers as possible. For more information, see Driver Verifier.

Bug Check 0x51: REGISTRY_ERROR

7/2/2019 • 2 minutes to read • Edit Online

The REGISTRY_ERROR bug check has a value of 0x00000051. This indicates that a severe registry error has occurred.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

REGISTRY_ERROR Parameters

PARAMETER	DESCRIPTION
1	Reserved
2	Reserved
3	The pointer to the hive (if available)
4	If the hive is corrupt, the return code of **HvCheckHive** (if available)

Cause

Something has gone wrong with the registry. If a kernel debugger is available, get a stack trace: the **!analyze** debug extension displays information about the bug check and can be very helpful in determining the root cause, then enter one of the **k (Display Stack Backtrace)** commands to view the call stack.

This error may indicate that the registry encountered an I/O error while trying to read one of its files. This can be caused by hardware problems or file system corruption.

It may also occur due to a failure in a refresh operation, which is used only in by the security system, and then only when resource limits are encountered.

Bug Check 0x52: MAILSLOT_FILE_SYSTEM

7/2/2019 • 2 minutes to read • Edit Online

The MAILSLOT_FILE_SYSTEM bug check has a value of 0x00000052.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Bug Check 0x53: NO_BOOT_DEVICE

7/2/2019 • 2 minutes to read • Edit Online

The NO_BOOT_DEVICE bug check has a value of 0x00000053.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Bug Check 0x54: LM_SERVER_INTERNAL_ERROR

7/2/2019 • 2 minutes to read • Edit Online

The LM_SERVER_INTERNAL_ERROR bug check has a value of 0x00000054.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Bug Check 0x55: DATA_COHERENCY_EXCEPTION

11/22/2019 • 2 minutes to read • Edit Online

The DATA_COHERENCY_EXCEPTION bug check has a value of 0x00000055.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Resolution

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause.

Bug Check 0x56: INSTRUCTION_COHERENCY_EXCEPTION

11/22/2019 • 2 minutes to read • Edit Online

The INSTRUCTION_COHERENCY_EXCEPTION bug check has a value of 0x00000056.

This bug check appears very infrequently.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Resolution

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause.

Bug Check 0x57: XNS_INTERNAL_ERROR

7/2/2019 • 2 minutes to read • Edit Online

The XNS_INTERNAL_ERROR bug check has a value of 0x00000057.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Bug Check 0x58: FTDISK_INTERNAL_ERROR

7/2/2019 • 2 minutes to read • Edit Online

The FTDISK_INTERNAL_ERROR bug check has a value of 0x00000058. This is issued if the system is booted from the wrong copy of a mirrored partition.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

FTDISK_INTERNAL_ERROR Parameters

None

Cause

The hives are indicating that the mirror is valid, but it is not. The hives should actually be pointing to the shadow partition.

This is almost always caused by the primary partition being revived.

Resolution

Reboot the system from the shadow partition.

Bug Check 0x59: PINBALL_FILE_SYSTEM

7/2/2019 • 2 minutes to read • Edit Online

The PINBALL_FILE_SYSTEM bug check has a value of 0x00000059. This indicates that a problem occurred in the Pinball file system.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

PINBALL_FILE_SYSTEM Parameters

PARAMETER	DESCRIPTION
1	Specifies source file and line number information. The high 16 bits (the first four hexadecimal digits after the "0x") identify the source file by its identifier number. The low 16 bits identify the source line in the file where the bug check occurred.
2	Reserved
3	Reserved
4	Reserved

Cause

One possible cause of this bug check is depletion of nonpaged pool memory. If the nonpaged pool memory is completely depleted, this error can stop the system. However, during the indexing process, if the amount of available nonpaged pool memory is very low, another kernel-mode driver requiring nonpaged pool memory can also trigger this error.

Resolution

To resolve a nonpaged pool memory depletion problem: Add new physical memory to the computer. This will increase the quantity of nonpaged pool memory available to the kernel.

Bug Check 0x5A: CRITICAL_SERVICE_FAILED

7/15/2019 • 2 minutes to read • Edit Online

The CRITICAL_SERVICE_FAILED bug check has a value of 0x0000005A.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Resolution

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause.

Bug Check 0x5B: SET_ENV_VAR_FAILED

7/2/2019 • 2 minutes to read • Edit Online

The SET_ENV_VAR_FAILED bug check has a value of 0x0000005B.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Bug Check 0x5C: HAL_INITIALIZATION_FAILED

7/2/2019 • 2 minutes to read • Edit Online

The HAL_INITIALIZATION_FAILED bug check has a value of 0x0000005C.

This indicates that the HAL initialization failed.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Bug Check 0x5D: UNSUPPORTED_PROCESSOR

7/2/2019 • 2 minutes to read • Edit Online

The UNSUPPORTED_PROCESSOR bug check has a value of 0x0000005D. This indicates that the computer is attempting to run Windows on an unsupported processor.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

UNSUPPORTED_PROCESSOR Parameters

None

Cause

Windows requires a higher-grade processor than the one you are using.

Bug Check 0x5E: OBJECT_INITIALIZATION_FAILED

7/2/2019 • 2 minutes to read • Edit Online

The OBJECT_INITIALIZATION_FAILED bug check has a value of 0x0000005E.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Bug Check 0x5F: SECURITY_INITIALIZATION_FAILED

7/2/2019 • 2 minutes to read • Edit Online

The SECURITY_INITIALIZATION_FAILED bug check has a value of 0x0000005F.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Bug Check 0x60: PROCESS_INITIALIZATION_FAILED

7/2/2019 • 2 minutes to read • Edit Online

The PROCESS_INITIALIZATION_FAILED bug check has a value of 0x00000060.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Resolution

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause.

Bug Check 0x61: HAL1_INITIALIZATION_FAILED

7/2/2019 • 2 minutes to read • Edit Online

The HAL1_INITIALIZATION_FAILED bug check has a value of 0x00000061.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Bug Check 0x62: OBJECT1_INITIALIZATION_FAILED

7/2/2019 • 2 minutes to read • Edit Online

The OBJECT1_INITIALIZATION_FAILED bug check has a value of 0x00000062.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Bug Check 0x63: SECURITY1_INITIALIZATION_FAILED

7/2/2019 • 2 minutes to read • Edit Online

The SECURITY1_INITIALIZATION_FAILED bug check has a value of 0x00000063.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Bug Check 0x64: SYMBOLIC_INITIALIZATION_FAILED

7/2/2019 • 2 minutes to read • Edit Online

The SYMBOLIC_INITIALIZATION_FAILED bug check has a value of 0x00000064.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Bug Check 0x65: MEMORY1_INITIALIZATION_FAILED

7/2/2019 • 2 minutes to read • Edit Online

The MEMORY1_INITIALIZATION_FAILED bug check has a value of 0x00000065.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Bug Check 0x66: CACHE_INITIALIZATION_FAILED

7/2/2019 • 2 minutes to read • Edit Online

The CACHE_INITIALIZATION_FAILED bug check has a value of 0x00000066.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Bug Check 0x67: CONFIG_INITIALIZATION_FAILED

7/2/2019 • 2 minutes to read • Edit Online

The CONFIG_INITIALIZATION_FAILED bug check has a value of 0x00000067. This bug check indicates that the registry configuration failed.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

CONFIG_INITIALIZATION_FAILED Parameters

PARAMETER	DESCRIPTION
1	Reserved
2	The location selector
3	The NT status code
4	Reserved

Cause

The registry could not allocate the pool that it needed to contain the registry files. This situation should never occur, because the register allocates this pool early enough in system initialization so that plenty of paged pool should be available.

Bug Check 0x68: FILE_INITIALIZATION_FAILED

7/2/2019 • 2 minutes to read • Edit Online

The FILE_INITIALIZATION_FAILED bug check has a value of 0x00000068.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Bug Check 0x69: IO1_INITIALIZATION_FAILED

7/2/2019 • 2 minutes to read • Edit Online

The IO1_INITIALIZATION_FAILED bug check has a value of 0x00000069. This bug check indicates that the initialization of the I/O system failed for some reason.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

IO1_INITIALIZATION_FAILED Parameters

None

Cause

There is very little information available to analyze this error.

Most likely, the setup routine has improperly installed the system, or a user has reconfigured the system.

Bug Check 0x6A: LPC_INITIALIZATION_FAILED

7/2/2019 • 2 minutes to read • Edit Online

The LPC_INITIALIZATION_FAILED bug check has a value of 0x0000006A.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Bug Check 0x6B: PROCESS1_INITIALIZATION_FAILED

7/2/2019 • 2 minutes to read • Edit Online

The PROCESS1_INITIALIZATION_FAILED bug check has a value of 0x0000006B. This bug check indicates that the initialization of the Microsoft Windows operating system failed.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

PROCESS1_INITIALIZATION_FAILED Parameters

The following parameters appear on the blue screen.

PARAMETER	DESCRIPTION
1	The NT status code that caused the failure
2	Reserved
3	Reserved
4	Reserved

Cause

Any part of the disk subsystem can cause the PROCESS1_INITIALIZATION_FAILED bug check, including bad disks, bad or incorrect cables, mixing different ATA-type devices on the same chain, or drives that are not available because of hardware regeneration.

This bug check can also be caused by a missing file from the boot partition or by a driver that a user accidentally disabled in the **Drivers** tab.

Resolution

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause.

Bug Check 0x6C: REFMON_INITIALIZATION_FAILED

7/2/2019 • 2 minutes to read • Edit Online

The REFMON_INITIALIZATION_FAILED bug check has a value of 0x0000006C.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Bug Check 0x6D: SESSION1_INITIALIZATION_FAILED

7/2/2019 • 2 minutes to read • Edit Online

The SESSION1_INITIALIZATION_FAILED bug check has a value of 0x0000006D. This bug check indicates that the initialization of the Microsoft Windows operating system failed.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

SESSION1_INITIALIZATION_FAILED Parameters

PARAMETER	DESCRIPTION
1	The NT status code that caused the initialization failure
2	0
3	0
4	0

Bug Check 0x6E: SESSION2_INITIALIZATION_FAILED

7/2/2019 • 2 minutes to read • Edit Online

The SESSION2_INITIALIZATION_FAILED bug check has a value of 0x0000006E. This bug check indicates that the initialization of the Microsoft Windows operating system failed.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

SESSION2_INITIALIZATION_FAILED Parameters

PARAMETER	DESCRIPTION
1	The NT status code that caused the Windows operating system to conclude that initialization failed
2	0
3	0
4	0

Bug Check 0x6F: SESSION3_INITIALIZATION_FAILED

7/2/2019 • 2 minutes to read • Edit Online

The SESSION3_INITIALIZATION_FAILED bug check has a value of 0x0000006F. This bug check indicates that the initialization of the Microsoft Windows operating system failed.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

SESSION3_INITIALIZATION_FAILED Parameters

PARAMETER	DESCRIPTION
1	The NT status code that caused the Windows operating system to conclude that initialization failed
2	0
3	0
4	0

Bug Check 0x70: SESSION4_INITIALIZATION_FAILED

7/2/2019 • 2 minutes to read • Edit Online

The SESSION4_INITIALIZATION_FAILED bug check has a value of 0x00000070. This bug check indicates that the initialization of the Microsoft Windows operating system failed.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

SESSION4_INITIALIZATION_FAILED Parameters

PARAMETER	DESCRIPTION
1	The NT status code that caused the Windows operating system to conclude that initialization failed
2	0
3	0
4	0

Bug Check 0x71: SESSION5_INITIALIZATION_FAILED

7/2/2019 • 2 minutes to read • Edit Online

The SESSION5_INITIALIZATION_FAILED bug check has a value of 0x00000071. This bug check indicates that the initialization of the Microsoft Windows operating system failed.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

SESSION5_INITIALIZATION_FAILED Parameters

PARAMETER	DESCRIPTION
1	The NT status code that caused the Windows operating system to conclude that initialization failed
2	0
3	0
4	0

Bug Check 0x72: ASSIGN_DRIVE_LETTERS_FAILED

7/2/2019 • 2 minutes to read • Edit Online

The ASSIGN_DRIVE_LETTERS_FAILED bug check has a value of 0x00000072.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Bug Check 0x73: CONFIG_LIST_FAILED

7/2/2019 • 2 minutes to read • Edit Online

The CONFIG_LIST_FAILED bug check has a value of 0x00000073. This bug check indicates that one of the top-level registry keys, also known as core system hives, cannot be linked in the registry tree.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

CONFIG_LIST_FAILED Parameters

PARAMETER	DESCRIPTION
1	1
2	The NT status code that led the Windows operating system to assume that it failed to load the hive
3	The index of the hive in the hive list
4	A pointer to a UNICODE_STRING structure that contains the file name of the hive

Cause

The registry hive that cannot be linked might be SAM, SECURITY, SOFTWARE, or DEFAULT. The hive is valid, because it was loaded successfully.

Examine Parameter 2 to see why the hive could not be linked in the registry tree. One common cause of this error is that the Windows operating system is out of disk space on the system drive. (In this situation, this parameter is 0xC000017D, STATUS_NO_LOG_SPACE.) Another common problem is that an attempt to allocate pool has failed. (In this situation, Parameter 2 is 0xC000009A, STATUS_INSUFFICIENT_RESOURCES.) You must investigate other status codes.

Bug Check 0x74: BAD_SYSTEM_CONFIG_INFO

8/26/2019 • 3 minutes to read • Edit Online

The BAD_SYSTEM_CONFIG_INFO bug check has a value of 0x00000074. This bug check indicates that there is an error in the registry.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

BAD_SYSTEM_CONFIG_INFO Parameters

PARAMETER	DESCRIPTION
1	Reserved
2	Reserved
3	Reserved
4	The NT status value/code (if it is available)

Cause

The BAD_SYSTEM_CONFIG_INFO bug check occurs if the SYSTEM hive is corrupt. However, this corruption is unlikely, because the boot loader, checks a hive for corruption when it loads the hive.

This bug check can also occur if some critical registry keys and values are missing. The keys and values might be missing if a user manually edited the registry or if an application or service corrupted the registry.

Looking up the NT status value returned in parameter 4 can provide additional information, see NTSTATUS Values for a listing.

Resolution

Check in the Windows system eventlog to see if there are any registry related error events. If there are see if the event lists a hive or specific key that the error occurred in.

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause.

```
BAD_SYSTEM_CONFIG_INFO (74)
Can indicate that the SYSTEM hive loaded by the osloader/NTLDR
was corrupt.  This is unlikely, since the osloader will check
a hive to make sure it isn't corrupt after loading it.
It can also indicate that some critical registry keys and values
are not present.  (i.e. somebody used regedt32 to delete something
that they shouldn't have)  Booting from LastKnownGood may fix
the problem, but if someone is persistent enough in mucking with
the registry they will need to reinstall or use the Emergency
Repair Disk.
Arguments:
Arg1: 0000000000000002, (reserved)
Arg2: ffffd481054b49f0, (reserved)
Arg3: 0000000000000004, (reserved)
Arg4: ffffffffc000014c, usually the NT status code.
```

Review all of the information returned by the !analyze to learn about the failure.

Use the !error extension to display information about the NTSTATUS value in parameter 4.

```
2: kd> !ERROR ffffffffc000014c
Error code: (NTSTATUS) 0xc000014c (3221225804) - {The Registry Is Corrupt}  The structure of one of the files
that contains Registry data is corrupt, or the image of the file in memory is corrupt, or the file could not
be recovered because the alternate copy or log was absent or corrupt.
```

Use the !reg extenstion to display information about the registry, for example the hives present in the registry.

```
!reg hivelist

---------------------------------------------------------------------------------------------------------------
---------------------------------------
|     HiveAddr     |Stable Length|    Stable Map    |Volatile Length|    Volatile Map
|MappedViews|PinnedViews|U(Cnt)|     BaseBlock     | FileName
---------------------------------------------------------------------------------------------------------------
---------------------------------------
| ffff95077ea24000 |       1000  | ffff95077ea24588 |          0    |  0000000000000000  |     0|
ffff95077ea31000  | <NONAME>
| ffff95077ea3e000 |    12d3000  | ffff95077ea49000 |      21000    |  ffff95077ea3e800  |     0|
ffff95077ea40000  | SYSTEM
| ffff95077ea8f000 |      53000  | ffff95077ea8f588 |       9000    |  ffff95077ea8f800  |     0|
ffff95077ea91000  | <NONAME>
| ffff9507821c8000 |       7000  | ffff9507821c8588 |          0    |  0000000000000000  |     0|
ffff9507821cc000  | kVolume2\EFI\Microsoft\Boot\BCD
| ffff95077f6ae000 |    685c000  | ffff95077f737000 |       6000    |  ffff95077f6ae800  |     0|
ffff95077f6b6000  | emRoot\System32\Config\SOFTWARE
---------------------------------------------------------------------------------------------------------------
---------------------------------------
```

Use the !reg openkeys command to see which registry keys were open.

```
2: kd> !reg openkeys

Hive: \REGISTRY\MACHINE\SYSTEM
================================================================================
Index 0:   00000000 kcb=ffffd805e303c728 cell=00000020 f=002c0100 \REGISTRY\MACHINE\SYSTEM
Index 1:   db67f96d kcb=ffffd805e416ed18 cell=00bd0b40 f=00200080 \REGISTRY\MACHINE\SYSTEM\WPA\8DEC0AF1-0341-
4B93-85CD-72606C2DF94C-7P-374
Index 3:   db67ee93 kcb=ffffd805e30c5ab8 cell=00bc1550 f=00200080 \REGISTRY\MACHINE\SYSTEM\WPA\8DEC0AF1-0341-
4B93-85CD-72606C2DF94C-7P-161
Index 4:   f9909d96 kcb=ffffd805e44bd268 cell=00bf8f50 f=00200000
\REGISTRY\MACHINE\SYSTEM\CONTROLSET001\CONTROL\POWER\PROFILE\EVENTS\{54533251-82BE-4824-96C1-47B60B740D00}\
{8BC6262C-C026-411D-AE3B-7E2F70811A13}
Index 5:   e9dd6ce5 kcb=ffffd805e4180e48 cell=00812970 f=00200000 \REGISTRY\MACHINE\SYSTEM\DRIVERDATABASE

...
```

Remarks

Try booting into safe mode and then restart the OS normally. If the restart does not fix the problem, the registry damage is too extensive. Try the following steps.

- If you have a system restore point, try restoring to an earlier restore point.
- Reset your PC.
- Use installation media to restore or reset your PC.
- Use installation media to reinstall Windows.

For more information, see Recovery options in Windows 10.

This support article discusses this bug check code: Error 0x74: Bad_system_config_info

Bug Check 0x75: CANNOT_WRITE_CONFIGURATION

7/2/2019 • 2 minutes to read • Edit Online

The CANNOT_WRITE_CONFIGURATION bug check has a value of 0x00000075. This bug check indicates that the SYSTEM registry hive file cannot be converted to a mapped file.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

CANNOT_WRITE_CONFIGURATION Parameters

PARAMETER	DESCRIPTION
1	1
2	The NT status code that led the Windows operating system to assume that it had failed to convert the hive
3	Reserved
4	Reserved

Cause

The CANNOT_WRITE_CONFIGURATION bug check typically occurs if the system is out of pool and the Windows operating system cannot reopen the hive.

This bug check should almost never occur, because the conversion of the hive file occurs early enough during system initialization so that enough pool should be available.

Bug Check 0x76: PROCESS_HAS_LOCKED_PAGES

7/2/2019 • 2 minutes to read • Edit Online

The PROCESS_HAS_LOCKED_PAGES bug check has a value of 0x00000076. This bug check indicates that a driver failed to release locked pages after an I/O operation, or that it attempted to unlock pages that were already unlocked.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

PROCESS_HAS_LOCKED_PAGES Parameters

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x00	The pointer to the process object	The number of locked pages	The pointer to driver stacks (if they are enabled). Otherwise, this parameter is zero.	The process being terminated has locked memory pages. The driver must unlock any memory that it might have locked in a process, before the process terminates.
0x01	MDL specified by the driver	Current number of locked memory pages in that process	A pointer to driver stacks for that process (if they are enabled). Otherwise, this parameter is zero.	The driver is attempting to unlock process memory pages that are not locked.

Cause

The driver either failed to unlock pages that it locked (parameter 1 value is 0x0), or the driver is attempting to unlock pages that have not been locked or that have already been unlocked (parameter 1 value is 0x1).

Resolution

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause.

If the parameter 1 value is 0x0

First use the **!search** extension on the current process pointer throughout all of physical memory. This extension might find at least one memory descriptor list (MDL) that points to the current process. Next, use **!search** on each MDL that you find to obtain the I/O request packet (IRP) that points to the current process. From this IRP, you can identify which driver is leaking the pages.

Otherwise, you can detect which driver caused the error by editing the registry:

1. In the **\\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management** registry key, create or edit the **TrackLockedPages** value, and then set it equal to DWORD 1.
2. Restart the computer.

The system then saves stack traces, so you can easily identify the driver that caused the problem. If the driver causes the same error again, **bug check 0xCB** (DRIVER_LEFT_LOCKED_PAGES_IN_PROCESS) is issued, and the name of the driver that causes this error is displayed on the blue screen and stored in memory at the location (PUNICODE_STRING) **KiBugCheckDriver**.

If the parameter 1 value is 0x1

Examine the driver source code that locks and unlocks memory, and try to locate an instance where memory is unlocked without first being locked.

Bug Check 0x77: KERNEL_STACK_INPAGE_ERROR

7/2/2019 • 4 minutes to read • Edit Online

The KERNEL_STACK_INPAGE_ERROR bug check has a value of 0x00000077. This bug check indicates that the requested page of kernel data from the paging file could not be read into memory.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

KERNEL_STACK_INPAGE_ERROR Parameters

The four parameters that listed in the message have two possible meanings.

If the first parameter is 0, 1, or 2, the parameters have the following meaning.

PARAMETER	DESCRIPTION
1	**0:** The page of kernel data was retrieved from page cache. **1:** The page was retrieved from a disk. **2:** The page was retrieved from a disk, the storage stack returned SUCCESS, but **Status.Information** is not equal to PAGE_SIZE.
2	The value that appears in the stack where the signature should be.
3	0
4	The address of the signature on the kernel stack

If the first parameter is any value other than 0, 1, or 2, the parameters have the following meaning.

PARAMETER	DESCRIPTION
1	The status code
2	The I/O status code
3	The page file number
4	The offset into page file

Cause

If the first parameter is 0 or 1, the stack signature in the kernel stack was not found. This error is probably caused by defective hardware, such as a RAM error.

If the first parameter is 2, the driver stack returned an inconsistent status for the read of the page. For example, the driver stack returned a success status even though it did not read the whole page.

If the first parameter is any value other than 0, 1, or 2, the value of the first parameter is an NTSTATUS error code that the driver stack returns after it tries to retrieve the page of kernel data. You can determine the exact cause of this error from the I/O status code (the second parameter). Some common status codes include the following:

- 0xC000009A, or STATUS_INSUFFICIENT_RESOURCES, indicates a lack of nonpaged pool resources. This status code indicates a driver error in the storage stack. (The storage stack should always be able to retrieve this data, regardless of software resource availability.)
- 0xC000009C, or STATUS_DEVICE_DATA_ERROR, indicates bad blocks (sectors) on the hard disk.
- 0xC000009D, or STATUS_DEVICE_NOT_CONNECTED, indicates defective or loose cabling, termination, or that the controller does not see the hard disk drive.
- 0xC000016A, or STATUS_DISK_OPERATION_FAILED, indicates bad blocks (sectors) on the hard disk.
- 0xC0000185, or STATUS_IO_DEVICE_ERROR, indicates improper termination or defective cabling on SCSI devices or that two devices are trying to use the same IRQ.

These status codes are the most common ones that have specific causes. For more information about other possible status codes that might be returned, see the Ntstatus.h file in the Microsoft Windows Driver Kit (WDK).

A virus infection can also cause this bug check.

Resolution

Resolving a bad block problem: If you can restart the computer after the error, Autochk runs automatically and attempts to map the bad sector to prevent it from being used anymore.

If Autochk does not scan the hard disk for errors, you can manually start the disk scanner. Run **Chkdsk /f /r** on the system partition. You must restart the computer before the disk scan begins. If you cannot start the system because the error, use the Recovery Console and run **Chkdsk /r**.

Warning If your system partition is formatted with the FAT file system, the long file names that the Windows operating system uses might be damaged if you use Scandisk or another MS-DOS-based hard disk tool to verify the integrity of your hard disk drive from MS-DOS. Always use the version of Chkdsk that matches your version of the Windows operating system.

Resolving a defective hardware problem: If the I/O status is 0xC0000185 and the paging file is on an SCSI disk, check the disk cabling and SCSI termination for problems.

Resolving a failing RAM problem: Run the hardware diagnostics that the system manufacturer supplies, especially the memory scanner. For more information about these procedures, see the owner's manual for your computer.

Check that all the adapter cards in the computer are properly seated. Use an ink eraser or an electrical contact treatment, available at electronics supply stores, to ensure adapter card contacts are clean.

Check the System Log in Event Viewer for additional error messages that might help identify the device that is causing the error. You can also disable memory caching of the BIOS to try to resolve this error.

Make sure that the latest Windows Service Pack is installed.

If the preceding steps fail to resolve the error, take the system motherboard to a repair facility for diagnostic testing. A crack, a scratched trace, or a defective component on the motherboard can cause this error.

Resolving a virus infection: Check your computer for viruses by using any up-to-date, commercial virus scanning software that examines the Master Boot Record of the hard disk. All Windows file systems can be infected by viruses.

See also

Bug Check 0x7A (KERNEL_DATA_INPAGE_ERROR)

Bug Check 0x78: PHASE0_EXCEPTION

7/2/2019 • 2 minutes to read • Edit Online

The PHASE0_EXCEPTION bug check has a value of 0x00000078.

This bug check occurs when an unexpected break is encountered during HAL initialization. This break can occur if you have set the **/break** parameter in your boot settings but have not enabled kernel debugging.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Bug Check 0x79: MISMATCHED_HAL

7/2/2019 • 2 minutes to read • Edit Online

The MISMATCHED_HAL bug check has a value of 0x00000079. This bug check indicates that the Hardware Abstraction Layer (HAL) revision level or configuration does not match that of the kernel or the computer.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

MISMATCHED_HAL Parameters

Parameter 1 indicates the type of mismatch.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE.
0x1	The major processor control block (PRCB) level of Ntoskrnl.exe.	The major PRCB level of Hal.dll.	Reserved	The PRCB release levels are mismatched. (Something is out of date.)
0x2	The build type of Ntoskrnl.exe.	The build type of Hal.dll.	Reserved	The build types are mismatched.
0x3	The size of the loader parameter extension.	The major version of the loader parameter extension.	The minor version of the loader parameter extension.	The loader (ntldr) and HAL versions are mismatched.

When Parameter 1 equals 0x2, the following build type codes are used:

- 0: Multiprocessor-enabled free build
- 1: Multiprocessor-enabled checked build
- 2: Single-processor free build
- 3: Single-processor checked build

Cause

The MISMATCHED_HAL bug check often occurs when a user manually updates Ntoskrnl.exe or Hal.dll.

The error can also indicate that one of those two files is out of date. Or the computer might erroneously have a multiprocessor HAL and a single-processor kernel installed, or vice versa.

The Ntoskrnl.exe kernel file is for single-processor systems and Ntkrnlmp.exe is for multiprocessor systems. However, these file names correspond to the files on the installation media.After you have installed the Windows operating system, the file is renamed to Ntoskrnl.exe, regardless of the source file that is used. The HAL file also

uses the name Hal.dll after installation, but there are several possible HAL files on the installation media. For more information, see "Installing the Checked Build" in the Windows Driver Kit (WDK).

Resolution

Restart the computer by using the product CD or the Windows Setup disks. At the Welcome screen, press F10 to start the Recovery Console. Use the **Copy** command to copy the correct HAL or kernel file from the original CD into the appropriate folder on the hard disk. The **Copy** command detects whether the file that you are copying is in the Microsoft compressed file format. If so, it automatically expands the file that is copied on the target drive.

Bug Check 0x7A: KERNEL_DATA_INPAGE_ERROR

7/2/2019 • 4 minutes to read • Edit Online

The KERNEL_DATA_INPAGE_ERROR bug check has a value of 0x0000007A. This bug check indicates that the requested page of kernel data from the paging file could not be read into memory.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

KERNEL_DATA_INPAGE_ERROR Parameters

The four parameters that are listed in the message can have three possible meanings. If the first parameter is 1 or 2, or 3 and the third parameter is 0, the parameters have the following definitions.

PARAMETER	DESCRIPTION
1	The lock type that was held (1, 2, or 3)
2	The error status (usually an I/O status code)
3	**If Lock Type is 1:** the current process **If Lock Type is 2 or 3:** 0
4	The virtual address that could not be paged into memory

If the first parameter is 3 (and the third parameter is nonzero) or 4, the parameters have the following definitions.

PARAMETER	DESCRIPTION
1	The lock type that was held (3 or 4)
2	The error status (typically an I/O status code)
3	The address of the InPageSupport structure
4	The faulting address

Otherwise, the parameters have the following definitions.

PARAMETER	DESCRIPTION
1	The address of the page table entry (PTE)
2	The error status (usually an I/O status code)
3	The PTE contents
4	The faulting address

Cause

Frequently, you can determine the cause of the KERNEL_DATA_INPAGE_ERROR bug check from the error status (Parameter 2). Some common status codes include the following:

- 0xC000009A, or STATUS_INSUFFICIENT_RESOURCES, indicates a lack of nonpaged pool resources.
- 0xC000009C, or STATUS_DEVICE_DATA_ERROR, typically indicates bad blocks (sectors) on the hard disk.
- 0xC000009D, or STATUS_DEVICE_NOT_CONNECTED, indicates defective or loose cabling, termination, or that the controller does not see the hard disk.
- 0xC000016A, or STATUS_DISK_OPERATION_FAILED, indicates bad blocks (sectors) on the hard disk.
- 0xC0000185, or STATUS_IO_DEVICE_ERROR, indicates improper termination or defective cabling on SCSI devices or that two devices are trying to use the same IRQ.
- 0xC000000E, or STATUS_NO_SUCH_DEVICE, indicates a hardware failure or an incorrect drive configuration. Check your cables and check the drive with the diagnostic utility available from your drive manufacturer. If you are using older PATA (IDE) drives, this status code can indicate an incorrect master/subordinate drive configuration.

These status codes are the most common ones that have specific causes. For more information about other possible status codes that can be returned, see the Ntstatus.h file in the Microsoft Windows Driver Kit (WDK).

Another common cause of this error message is defective hardware or failing RAM.

A virus infection can also cause this bug check.

Resolution

Resolving a bad block problem: An I/O status code of 0xC000009C or 0xC000016A typically indicates that the data could not be read from the disk because of a bad block (sector). If you can restart the computer after the error, Autochk runs automatically and attempts to map the bad sector to prevent it from being used anymore.

If Autochk does not scan the hard disk for errors, you can manually start the disk scanner. Run **Chkdsk /f /r** on the system partition. You must restart the computer before the disk scan begins. If you cannot start the computer because of the error, use the Recovery Console and run **Chkdsk /r**.

Warning If your system partition is formatted with the FAT file system, the long file names that the Windows operating system uses might be damaged if you use Scandisk or another MS-DOS-based hard disk tool to verify the integrity of your hard disk from MS-DOS. Always use the version of Chkdsk that matches your version of Windows.

Resolving a defective hardware problem: If the I/O status is C0000185 and the paging file is on an SCSI disk, check the disk cabling and SCSI termination for problems.

Resolving a failing RAM problem: Run the hardware diagnostics that the system manufacturer supplies, especially the memory scanner. For more information about these procedures, see the owner's manual for your computer.

Check that all the adapter cards in the computer are properly seated. Use an ink eraser or an electrical contact treatment, available at electronics supply stores, to ensure adapter card contacts are clean.

Check the System Log in Event Viewer for additional error messages that might help identify the device that is causing the error. You can also disable memory caching of the BIOS to try to resolve this error.

Make sure that the latest Windows Service Pack is installed.

If the preceding steps do not resolve the error, take the system motherboard to a repair facility for diagnostic testing. A crack, a scratched trace, or a defective component on the motherboard can cause this error.

Resolving a virus infection: Check your computer for viruses by using any up-to-date, commercial virus scanning software that examines the Master Boot Record of the hard disk. All Windows file systems can be infected by viruses.

See also

Bug Check 0x77 (KERNEL_STACK_INPAGE_ERROR)

Bug Check 0x7B: INACCESSIBLE_BOOT_DEVICE

7/2/2019 • 9 minutes to read • Edit Online

The INACCESSIBLE_BOOT_DEVICE bug check has a value of 0x0000007B. This bug check indicates that the Microsoft Windows operating system has lost access to the system partition during startup.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

INACCESSIBLE_BOOT_DEVICE Parameters

PARAMETER	DESCRIPTION
1	The address of a UNICODE_STRING structure, or the address of the device object that could not be mounted
2	0
3	0
4	0

To determine the meaning of Parameter 1, look at the data that it points to. If the first word (USHORT) at this address is even, Parameter 1 is the beginning of a Unicode string. If the first word (USHORT) at this address is 0x3, Parameter 1 is the first field (Type) of a device object.

- If this parameter points to a device object, the file system that was supposed to read the boot device failed to initialize or simply did not recognize the data on the boot device as a file system structure. In this situation, the specified device object is the object that could not be mounted.
- If this parameter points to a Unicode string, you must read the first 8 bytes at this address. These bytes form the UNICODE_STRING structure, which is defined as follows:

  ```
  USHORT Length;
  USHORT MaximumLength;
  PWSTR Buffer;
  ```

  The **Length** field gives the actual length of the string. The **Buffer** field points to the beginning of the string (**Buffer** is always be at least 0x80000000.)

  The actual string contains the Advanced RISC Computing (ARC) specification name of the device that the boot was being attempted from. ARC names are a generic way to identify devices in the ARC environment.

Cause

The INACCESSIBLE_BOOT_DEVICE bug check frequently occurs because of a boot device failure. During I/O system initialization, the boot device driver might have failed to initialize the boot device (typically a hard disk).

File system initialization might have failed because it did not recognize the data on the boot device. Also, repartitioning the system partition, changing the BIOS configuration, or installing a disk controller might induce this error.

This error can also occur because of incompatible disk hardware. If the error occurred at the initial setup of the system, the system might have been installed on an unsupported disk controller. Some disk controllers require additional drivers to be present when Windows starts.

Resolution

This error always occurs while the system is starting. This error frequently occurs before the debugger connection is established, so debugging can be difficult. In addition, the OS may not be accessible and the error logs may be empty as the OS has not booted far enough to start those sub-systems.

****

If you are unable to boot Windows

****

If you receive this stop code and Windows doesn't boot forward into the OS, try the following:

Revert any recent hardware changes

Remove any recently added hardware, especially hard disk drives or controllers, to see if the error is resolved. If the problematic hardware is a hard disk drive, the disk firmware version might be incompatible with your version of the Windows operating system. Contact the manufacturer for updates. If you removed another piece of hardware and the error is resolved, IRQ or I/O port conflicts may exist. Reconfigure the new device according to the manufacturer's instructions.

If you have recently made changes to BIOS settings, such as changing the controller mode from legacy to AHCI in the BIOS, revert those changes. For more information, see https://en.wikipedia.org/wiki/Advanced_Host_Controller_Interface

Check for storage device compatibility

Confirm that all hard disk drivers, hard disk controllers, and any other storage adapters are compatible with the installed version of Windows. For example, you can get information about compatibility at Windows 10 Specifications.

Update BIOS and Firmware

Check the availability of updates for the system BIOS and storage controller firmware.

Use the Media Creation Tool to create a bootable USB thumb drive or DVD

Use Media Creation Tool, using another computer to create a bootable USB thumb drive or DVD. Use it to perform a clean install, by clicking on the setup file or booting from the USB.

For more information, see Get Windows 10.

Be aware that you may need to disable features, such as quick BIOS boot, or you may not be able to reach the boot device priority menu. Change your boot sequence priority in the BIOS menu to boot from FDD (FlashDiskDrive) or DVD instead of HDD.

Common Boot Menu Keys

The boot menu key varies per manufacturer, these keys are commonly used. Check the system documentation to determine what boot key is used.

F12 ESC F9 F10 F8 **Common BIOS Setup Keys**

The BIOS setup key varies per manufacturer, these keys are commonly used. Check the system documentation to determine what setup key is used.

ESC DEL F2 ***********

If you can boot Windows

****

- **Boot to Safe Mode and then Boot Normally**

  Complete the following steps to boot into Safe Mode. Booting into safe mode loads a core set of storage drivers that may allow for the storage system to be accessed once again.

  To enter Safe Mode, use **Update and Security** in Settings. Select **Recovery**->**Advanced startup** to boot to maintenance mode. At the resulting menu, choose **Troubleshoot**-> **Advanced Options** -> **Startup Settings** -> **Restart**. After Windows restarts to the **Startup Settings** screen, select option, 4, 5 or 6 to boot to Safe Mode.

  Once Windows is loaded in Safe Mode, restart your PC to see if the proper storage drivers will be loaded and that the storage device is recognized.

  Safe Mode may also be available by pressing a function key on boot, for example F8. Refer to information from the system manufacturer for specific startup options.

- Use the scan disk utility to confirm that there are no file system errors. Right click on the drive you want to scan and select **Properties**. Click on **Tools**. Click the **Check now** button.

- Run a virus detection program. Viruses can infect all types of hard disks formatted for Windows, and resulting disk corruption can generate system bug check codes. Make sure the virus detection program checks the Master Boot Record for infections.

- For IDE devices, define the onboard IDE port as Primary only. Also check each IDE device for the proper **master/subordinate/stand alone** setting. Try removing all IDE devices except for hard disks. Finally, check the System Log in Event Viewer for additional error messages that might help identify the device or driver that is causing the error.

- Confirm that there is sufficient free space on the hard drive. The operating system and some applications require sufficient free space to create swap files and for other functions. Based on the system configuration, the exact requirement varies, but it is normally a good idea to have 10% to 15% free space available.

- Look in **Device Manager** to see if any devices are marked with the exclamation point (!). Review the events log displayed in driver properties for any faulting driver. Try updating the related driver.

- Check the System Log in Event Viewer for additional error messages that might help pinpoint the device or driver that is causing the error. For more information, see Open Event Viewer. Look for critical errors in the system log that occurred in the same time window as the blue screen.

- You can try running the hardware diagnostics supplied by the system manufacturer.

- Use the System File Checker tool to repair missing or corrupted system files. The System File Checker is a utility in Windows that allows users to scan for corruptions in Windows system files and restore corrupted files. Use the following command to run the System File Checker tool (SFC.exe).

```
SFC /scannow
```

For more information, see Use the System File Checker tool to repair missing or corrupted system files.

- After automatic repair, on the Choose an option screen, select **Troubleshoot > Advanced options > System Restore**. This option takes your PC back to an earlier point in time, called a system restore point. Restore points are generated when you install a new app, driver, update, or when you create a restore point manually. Choose a restore point before you experienced the error.
- Use the kernel debugger to attach to the system and further analyze the failure as described in remarks.

Remarks

Investigating the storage system configuration

It is helpful to know as much as possible about the boot device that Windows is installed on. For example, you can investigate the following items:

- Find out what type of controller the boot device is connected to (SATA, IDE, etc). If you can boot the system, you can use device manager to examine the controller and disk driver properties and see the associated driver file as well as error events.
- Indicate if other devices are attached to the same controller that the boot device is on (SSD, DVD, and so on).
- Note the file system that is used on the drive, typically NTFS.

To analyze this error using the kernel debugger: Run an **lm (List Loaded Modules)** command in the debugger to see which modules are loaded to attempt to isolate the specific driver. Verify that the following drivers were loaded.

disk

```
0: kd> lm m disk
Browse full module list
start             end                 module name
fffff806`bd0b0000 fffff806`bd0cd000   disk       (deferred)
```

partmgr

```
0: kd> lm m partmgr
Browse full module list
start             end                 module name
fffff806`bc5a0000 fffff806`bc5c1000   partmgr    (deferred)
```

NTFS

```
0: kd> lm m ntfs
Browse full module list
start             end                 module name
fffff806`bd3f0000 fffff806`bd607000   NTFS       (deferred)
```

classpnp

```
 0: kd> lm m classpnp
Browse full module list
start             end                 module name
fffff806`bd0d0000 fffff806`bd131000   CLASSPNP   (deferred)
```

pci

```
0: kd> lm m pci
Browse full module list
start             end                 module name
fffff806`bc440000 fffff806`bc494000   pci        (deferred)
```

Also make sure your controller drivers are loaded. For example for a SATA RAID Controller, this might be the *iaStorA.Sys* driver, or it could be the *EhStorClass* driver.

```
0: kd> lm m EhStorClass
Browse full module list
start             end                 module name
fffff806`bcbb0000 fffff806`bcbcb000   EhStorClass   (deferred)
```

List the drivers that contain "stor", storahci, may be present.

```
0: kd> lm m stor*
Browse full module list
start             end                 module name
fffff806`bcb00000 fffff806`bcb23000   storahci   (deferred)
fffff806`bcb30000 fffff806`bcbaa000   storport   (deferred)
fffff806`c0770000 fffff806`c0788000   storqosflt   (deferred)
```

Booting with a debugger attached

If you can boot the target system with a debugger connected, issue **!devnode 0 1** when the bugcheck occurs. You'll see which device lacks a driver or could not start, and the reason for not starting may be apparent.

One cause, might be that Plug and Play cannot assign resources to the boot device. You can verify this restriction by finding an entry for the service. If the status flags include DNF_INSUFFICIENT_RESOURCES or do not include DNF_STARTED or DNF_ENUMERATED, you may have located the problem. Try **!devnode 0 1 storahci** to save some time, instead of dumping the whole device tree.

```
0: kd> !devnode 0 1 storahci
Dumping IopRootDeviceNode (= 0xffffb9053d94d850)
DevNode 0xffffb9053e8dea50 for PDO 0xffffb9053e8da060
  InstancePath is "PCI\VEN_8086&DEV_3B22&SUBSYS_304A103C&REV_05\3&21436425&0&FA"
  ServiceName is "storahci"
  State = DeviceNodeStarted (0x308)
  Previous State = DeviceNodeEnumerateCompletion (0x30d)
  DevNode 0xffffb9053e88db30 for PDO 0xffffb9053e890060
    InstancePath is "SCSI\Disk&Ven_&Prod_ST3500418AS\4&23d99fa2&0&000000"
    ServiceName is "disk"
    State = DeviceNodeStarted (0x308)
    Previous State = DeviceNodeEnumerateCompletion (0x30d)
  DevNode 0xffffb9053e88d850 for PDO 0xffffb9053e88e060
    InstancePath is "SCSI\CdRom&Ven_hp&Prod_DVD-RAM_GH60L\4&23d99fa2&0&010000"
    ServiceName is "cdrom"
    TargetDeviceNotify List - f 0xffffdf0ae9bbb0e0  b 0xffffdf0aea874710
    State = DeviceNodeStarted (0x308)
    Previous State = DeviceNodeEnumerateCompletion (0x30d)
```

Bug Check 0x7C: BUGCODE_NDIS_DRIVER

10/23/2019 • 13 minutes to read • Edit Online

The BUGCODE_NDIS_DRIVER bug check has a value of 0x0000007C. This bug check indicates that the operating system detected an error in a networking driver.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

BUGCODE_NDIS_DRIVER Parameters

Parameter 1 indicates the type of violation. The meaning of the other parameters depends on the value of Parameter 1. If a Parameter's value is "0," that means it is not used.

PARAMETER 1	PARAMETER 1 VALUE AND CAUSE OF ERROR	PARAMETER 2	PARAMETER 3	PARAMETER 4
0x01	NDIS_BUGCHECK_ALLOCATE_SHARED_MEM_HIGH_IRQL A driver called **NdisMAllocateSharedMemory** at a raised IRQL.	The address of the specific miniport adapter block. Run **!ndiskd.netadapter** with this address for more information.	The length of the requested shared memory	The current IRQL
0x02	NDIS_BUGCHECK_SHARED_MEM_CORRUPTION During a call to **NdisMAllocateSharedMemory**, NDIS detected that a previously-allocated shared memory page had been corrupted.	The address of the specific miniport adapter block. Run **!ndiskd.netadapter** with this address for more information.	The shared memory page that was corrupted	The address of a NDIS_WRAPPER_CONTEXTE that keeps track of shared memory allocations by the driver

PARAMETER 1	PARAMETER 1 VALUE AND CAUSE OF ERROR	PARAMETER 2	PARAMETER 3	PARAMETER 4
0x03	NDIS_BUGCHECK_FREE_INVALID_SHARED_MEM A miniport driver called **NdisMFreeSharedMemory** (**Async**) with a shared memory address that had already been freed.	The address of the specific miniport adapter block. Run **!ndiskd.netadapter** with this address for more information.	The page from which this shared memory was allocated	The virtual address of the shared memory
0x04	NDIS_BUGCHECK_UNLOAD_DRIVER_INVALID_PARAMETER **AddDevice** was called with a driver that is not on the list of drivers that are registered with NDIS. Enabled only on special instrumented NDIS.	The address of the NDIS_M_DRIVER_BLOCK	The address of the DRIVER_OBJECT	0
0x05	NDIS_BUGCHECK_RECVD_PACKET_IN_USE_BAD_STACK_LOCATION An Ethernet driver indicated that it received a packet using a packet descriptor that was currently in use by the protocol stack. Caught by checking stack packet location.	The address of the specific miniport adapter block. Run **!ndiskd.netadapter** with this address for more information.	The address of the packet descriptor used by the driver. Run **!ndiskd.pkt** with this address for more information.	The address of the packet array that contained this packet descriptor

PARAMETER 1	PARAMETER 1 VALUE AND CAUSE OF ERROR	PARAMETER 2	PARAMETER 3	PARAMETER 4
0x06	NDIS_BUGCHECK_RECVD_PACKET_IN_USE_BAD_REF_COUNT An Ethernet driver indicated that it received a packet using a packet descriptor that was currently in use by the protocol stack. Caught by checking packet reference count.	The address of the specific miniport adapter block. Run **!ndiskd.netadapter** with this address for more information.	The address of the packet descriptor used by the driver. Run **!ndiskd.pkt** with this address for more information.	The address of the packet array that contained this packet descriptor
0x07	An FDDI driver indicated that it received a packet by using a packet descriptor that was currently in use by the protocol stack. Caught by checking reference count.	The address of the specific miniport adapter block. Run **!ndiskd.netadapter** with this address for more information.	The address of the packet descriptor used by the driver. Run **!ndiskd.pkt** with this address for more information.	The address of the packet array that contained this packet descriptor
0x08	NDIS_BUGCHECK_HALT_WITHOUT_INTERRUPT_DEREGISTER A miniport driver did not deregister its interrupt during the halt process.	The address of the specific miniport adapter block. Run **!ndiskd.netadapter** with this address for more information.	The address of the NDIS_MINIPORT_INTERRUPT	0
0x09	NDIS_BUGCHECK_HALT_WITHOUT_CANCEL_TIMER A miniport driver stopped without successfully canceling all its timers.	The address of the specific miniport adapter block. Run **!ndiskd.netadapter** with this address for more information.	The address of the miniport driver's timer queue (NDIS_MINIPORT_TIMER)	0

PARAMETER 1	PARAMETER 1 VALUE AND CAUSE OF ERROR	PARAMETER 2	PARAMETER 3	PARAMETER 4
0x0A	NDIS_BUGCHECK_DRIVER_UNLOAD_UNEXPECTED A miniport driver is getting unloaded prematurely.	The address of the NDIS_M_DRIVER_BLOCK	The address of the DRIVER_OBJECT	The reference count for the miniport driver
0x0B	NDIS_BUGCHECK_INIT_FAILED_WITHOUT_INTERRUPT_DEREGISTER A miniport driver failed its initialization without deregistering its interrupt.	The address of the specific miniport adapter block. Run **!ndiskd.netadapter** with this address for more information.	The address of the NDIS_MINIPORT_INTERRUPT	0
0x0C	NDIS_BUGCHECK_INIT_FAILED_WITHOUT_CANCEL_TIMER A miniport driver failed its initialization without successfully canceling all its timers.	The address of the specific miniport adapter block. Run **!ndiskd.netadapter** with this address for more information.	The address of the miniport driver's timer queue (NDIS_MINIPORT_TIMER)	0
0x0D	NDIS_BUGCHECK_HALT_INIT_WITHOUT_INTERRUPT_DEREGISTER A miniport driver did not deregister its interrupt during the halt process. The halt was called from the initialize routine after the miniport driver returned success from its initialize handler.	The address of the specific miniport adapter block. Run **!ndiskd.netadapter** with this address for more information.	The address of the NDIS_MINIPORT_INTERRUPT	0

PARAMETER 1	PARAMETER 1 VALUE AND CAUSE OF ERROR	PARAMETER 2	PARAMETER 3	PARAMETER 4
0x0E	NDIS_BUGCHECK_HALT_INIT_WITHOUT_CANCEL_TIMER A miniport driver stopped without successfully canceling all its timers. The halt was called from the initialize routine after the miniport driver returned success from its initialize handler.	The address of the specific miniport adapter block. Run **!ndiskd.netadapter** with this address for more information.	The address of the miniport driver's timer queue (NDIS_MINIPORT_TIMER)	0
0x0F	NDIS_BUGCHECK_RESET_COMPLETE_UNEXPECTED A miniport driver called **NdisMResetComplete** without any pending reset request.	The address of the specific miniport adapter block. Run **!ndiskd.netadapter** with this address for more information.	The reset status	AddressingReset (BOOLEAN)
0x10	NDIS_BUGCHECK_PM_INIT_FAILED_NO_INT_DEREGISTER After resuming from a low-power state, a miniport driver failed its initialization without deregistering its interrupt.	The address of the specific miniport adapter block. Run **!ndiskd.netadapter** with this address for more information.	The address of the NDIS_MINIPORT_INTERRUPT	0
0x11	NDIS_BUGCHECK_PM_INIT_FAILED_NO_CANCEL_TIMER After resuming from a low-power state, a miniport driver failed its initialization without successfully canceling all its timers.	The address of the specific miniport adapter block. Run **!ndiskd.netadapter** with this address for more information.	The address of the miniport driver's timer queue (NDIS_MINIPORT_TIMER)	0

PARAMETER 1	PARAMETER 1 VALUE AND CAUSE OF ERROR	PARAMETER 2	PARAMETER 3	PARAMETER 4
0x12	NDIS_BUGCHECK_NFILTER_RECVD_PACKET_BAD_REF_COUNT A miniport driver indicated that it received a packet using a packet descriptor that was currently in use by the protocol stack. Caught by checking packet reference count.	The address of the specific miniport adapter block. Run **!ndiskd.netadapter** with this address for more information.	The address of the packet descriptor used by the driver. Run **!ndiskd.pkt** with this address for more information.	The address of the packet array that contained this packet descriptor
0x13	NDIS_BUGCHECK_TFILTER_RECVD_PACKET_BAD_REF_COUNT A Token-Ring miniport driver indicated that it received a packet using a packet descriptor that was currently in use by the protocol stack. Caught by checking packet reference count.	The address of the specific miniport adapter block. Run **!ndiskd.netadapter** with this address for more information.	The address of the packet descriptor used by the driver. Run **!ndiskd.pkt** with this address for more information.	The address of the packet array that contained this packet descriptor
0x14	NDIS_BUGCHECK_WAIT_EVENT_HIGH_IRQL An NDIS driver called **NdisWaitEvent** at an illegal IRQL	The actual IRQL	0	0
0x15	NDIS_BUGCHECK_INVALID_NDIS5_CALL A miniport driver called an API that is reserved for older drivers. The driver should only call NDIS 6.x APIs.	The address of the specific miniport adapter block. Run **!ndiskd.netadapter** with this address for more information.	0	0

PARAMETER 1	PARAMETER 1 VALUE AND CAUSE OF ERROR	PARAMETER 2	PARAMETER 3	PARAMETER 4
0x16	NDIS_BUGCHECK_INVALID_OPEN_IN_BIND_CONTEXT A protocol driver improperly opened an adapter during binding.	The address of the specific protocol. Run **!ndiskd.protocol** with this address for more information.	The address of the context area that is allocated by the protocol driver. Cast to ndis!NDIS_BIND_CONTEXT.	The address of the open handle. Run **!ndiskd.mopen** with this address for more information.
0x17	NDIS_BUGCHECK_IFPROVIDER_DEREGISTER_UNEXPECTED An Interface Provider called **NdisIfDeregisterProvider** without first removing all its Interfaces.	The address of the interface provider handle. Run **!ndiskd.ifprovider** with this address for more information.	0	0
0x1B	NDIS_BUGCHECK_IF_STACK_TABLE_LOOP A driver attempted to add an Interface to the ifStackTable, but doing so would cause a cycle. The ifStackTable must not have cycles. Run **!ndiskd.ifstacktable** to see the current table (prior to this call to **NdisIfAddIfStackEntry**).	The HigherLayerIfIndex being added to the table	The LowerLayerIfIndex being added to the table	0
0x1C	NDIS_BUGCHECK_MINIPORT_FAILED_OID_WHICH_MUST_SUCCEED A miniport driver failed an OID request that must not fail. Doing so would leak memory or other resources.	The address of the specific miniport adapter block. Run **!ndiskd.netadapter** with this address for more information.	The OID that was failed. Use **!ndiskd.help** to find the name of this OID.	The failure status code (NDIS_STATUS_XXX) with which the OID request was completed

PARAMETER 1	PARAMETER 1 VALUE AND CAUSE OF ERROR	PARAMETER 2	PARAMETER 3	PARAMETER 4
0x1D	NDIS_BUGCHECK_OID_REQUEST_INVALID_BUFFER A miniport driver or filter driver has completed an OID request illegally. Check that BytesWritten is not greater than the entire length of the buffer.	The address of the specific miniport adapter or filter module block. Run **!ndiskd.netadapter** or **!ndiskd.filter** with this address for more information.	The address to the **NDIS_OID_REQUEST** that was completed illegally. Inspect it with **!ndiskd.oid**.	0
0x1E	NDIS_BUGCHECK_REFCOUNT_IMBALANCE NDIS has detected an error in an internal refcount. This can be caused by a refcount underflow (more dereferences than references), or by a tag mismatch.	0	Internal handle. Use **!ndiskd.ndisref** or cast to ndis!NDIS_REFCOUNT_BLOCK.	The current reftag value

PARAMETER 1	PARAMETER 1 VALUE AND CAUSE OF ERROR	PARAMETER 2	PARAMETER 3	PARAMETER 4
0x1F	NDIS_BUGCHECK_ILLEGAL_MINIPORT_STATE_TRANSITION A miniport driver completed a state transition illegally.	What failed. Possible values: 1. NDIS_BUGCHECK_ILLEGAL_MINIPORT_STATE_TRANSITION_PAUSE_COMPLETE The miniport called **NdisMPauseComplete** but there was no pending Pause operation. 2. NDIS_BUGCHECK_ILLEGAL_MINIPORT_STATE_TRANSITION_RESTART_COMPLETE The miniport called **NdisMRestartComplete** but there was no pending Restart operation.	The address of the specific miniport adapter block. Run **!ndiskd.netadapter** with this address for more information.	0
0x20	NDIS_BUGCHECK_STATUS_INDICATION_INVALID_BUFFER A miniport driver or filter driver indicated an illegal **NDIS_STATUS_INDICATION**.	The type of the status indication. Run **!ndiskd.help** with this code for more information.	The handle of the driver instance that indicated this illegal status indication. Run **!ndiskd.netadapter** or **!ndiskd.filter** with this handle for more information.	The address of the status indication payload. Its interpretation depends on the type of status indication.

PARAMETER 1	PARAMETER 1 VALUE AND CAUSE OF ERROR	PARAMETER 2	PARAMETER 3	PARAMETER 4
0x21	NDIS_BUGCHECK_INVALID_OBJECT_HEADER A driver created an invalid **NDIS_OBJECT_HEADER**.	The handle of the driver that indicated the illegal status indication. Run **!ndiskd.minidriver** or **!ndiskd.filterdriver** with this handle for more information.	The object with the malformed header. Its interpretation depends on the API being called. For example, if the driver called **NdisAllocateCloneOidRequest**, then cast the object to ndis!NDIS_OID_REQUEST.	0
0x22	NDIS_BUGCHECK_ILLEGAL_NET_PNP_EVENT A miniport driver or filter driver indicated an illegal **NET_PNP_EVENT_NOTIFICATION**.	The handle of the driver that indicated the illegal status indication. Run **!ndiskd.minidriver** or **!ndiskd.filterdriver** with this handle for more information.	Cast to NET_PNP_EVENT_NOTIFICATION	0
0x23	NDIS_BUGCHECK_PD_ERROR An error was detected in the Packet Direct datapath.	The subtype of the bugcheck. Possible values: 1. NDIS_BUGCHECK_PD_ERROR_EC_THREAD_MISMATCH An API was called on the wrong thread. 2. NDIS_BUGCHECK_PD_ERROR_ILLEGAL_ARM_BY_CLIENT A PD client attempted to arm the provider while in an illegal state. 3. NDIS_BUG	The value of Parameter 3 depends on the value of Parameter 2. Each number in this list corresponds to the same number in Parameter 2. 1. Cast to NDIS_PD_EC 2. Cast to NDIS_PD_QUEUE_TRACKER 3. Cast to NDIS_PD_QUEUE_TRACKER 4. Cast to NDIS_PD_QUEUE_TRACKER 5. The handle of the specific filter module. Run **!ndiskd.filter** with this handle for more information. 6. The buffer	The value of Parameter 4 depends on the value of Parameter 2. Each number in this list corresponds to the same number in Parameter 2. 1. The ETHREAD that was expected 2. The handle to the PD client 3. The handle to the PD provider. Run **!ndiskd.netadapter** with this handle for more information. 4. The handle to the PD provider. Run **!ndiskd.netadapter** with this handle for more information. 5. The handle to

PARAMETER 1	PARAMETER 1 VALUE AND CAUSE OF ERROR	PARAMETER 2	PARAMETER 3	PARAMETER 4
		CHECK_PD_ERROR_ILLEGAL_ARM_NOTIFICATION A PD provider illegally triggered a drain notification while it was not armed. 4. NDIS_BUGCHECK_PD_ERROR_ILLEGAL_ARM_NOTIFICATION_VIA_ISR A PD provider illegally triggered an ISR drain notification while it was not armed. 5. NDIS_BUGCHECK_PD_ERROR_ILLEGAL_THUNK_BY_LWF A filter driver attempted to interfere with the Packet Direct datapath. 6. NDIS_BUGCHECK_PD_ERROR_ILLEGAL_BM_GROUP_REQUEST A PD provider illegally attempted to remove itself from a buffer manager	manager group, if known 7. The source PD_MEMORY_HANDLE or PD_BUFFER	the PD provider. Run **!ndiskd.netadapter** with this handle for more information. 6. If Parameter 3 is 0, this is the provider handle. If Parameter 3 is non-zero, the PD client has not freed all allocations yet, and this is the PD client handle. 7. The target PD_BUFFER

PARAMETER 1	PARAMETER 1 VALUE AND CAUSE OF ERROR	PARAMETER 2	PARAMETER 3	PARAMETER 4
		group. 7 : NDIS_BUGCHECK_PD_ERROR_ILLEGAL_PD_BUFFER_SETUP A PD buffer setup request was malformed.		
0x24	NDIS_BUGCHECK_UNEXPECTED_FAILURE An internal operation failed unexpectedly. This is likely to be a bug in NDIS.SYS itself.	The operation that failed. Possible values: 0x01 : NDIS_BUGCHECK_UNEXPECTED_FAILURE_KEWAITFORSINGLEOBJECT A call to KeWaitForSingleObject failed.	The failure status code	0
0x25	NDIS_BUGCHECK_WATCHDOG An attempt to manage the network stack has taken too long. When NDIS calls out into other drivers, NDIS starts a watchdog timer to ensure the call completes promptly. If the call takes too long, NDIS injects a bugcheck. This can be caused by a simple deadlock. Look with "!stacks 2 ndis" or similar to see if any threads look suspicious. Pay special attention to the PrimaryThread from the NDIS_WATCHDOG_TRIAGE_BLOCK.	The operation that took too long. Possible values: • 0x01 : NDIS_BUGCHECK_WATCHDOG_PROTOCOL_PAUSE There was a timeout while pausing a protocol driver. • 0x02 : NDIS_BUGCHECK_WATCHDOG_PROTOCOL_NETPNPEVENT There was a timeout while delivering a NET_PNP_EVENT_N	Cast to ndis!NDIS_WATCHDOG_TRIAGE_BLOCK. Useful fields: • **StartTime** shows what time the operation started, in 100ns units, as returned by KeQueryInterruptTime. • **TimeoutMilliseconds** shows how long NDIS waited, at a minimum, before triggering this bugcheck. • **TargetObject** is a handle to the protocol, filter module, or miniport adapter that NDIS is waiting on. Run	The value of Parameter 4 depends on the value of Parameter 2. Each number in this list corresponds to the same hexadecimal value in Parameter 2. • 0x01 : 0 • 0x02 : The NET_PNP_EVENT_CODE of the stuck event. For more information about these codes, see **NET_PNP_EVENT**.. • 0x03 : The NDIS_STATUS code of the stuck indication. Use **!ndiskd.help** to decode it. • 0x04 : 0 • 0x11 : 0

PARAMETER 1	PARAMETER 1 VALUE AND CAUSE OF ERROR	PARAMETER 2	PARAMETER 3	PARAMETER 4
	This can be caused by lost NBLs, in which case **!ndiskd.pendingnbls** may help. Check for OIDs that are stuck using **!ndiskd.oid**.	OTIFICATION to a protocol driver. • 0x03 : NDIS_BUGCHECK_WATCHDOG_PROTOCOL_STATUS_INDICATION There was a timeout while delivering a status indication to a protocol driver. • 0x04 : NDIS_BUGCHECK_WATCHDOG_PROTOCOL_UNBIND There was a timeout while unbinding a protocol driver. • 0x11 : NDIS_BUGCHECK_WATCHDOG_FILTER_PAUSE There was a timeout while pausing a filter driver. • 0x12 : NDIS_BUGCHECK_WATCHDOG_FILTER_NETPNPEVENT There was a timeout while delivering a NET_PNP_EVENT_N	**!ndiskd.protocol**, **!ndiskd.filter**, or **!ndiskd.netadapter** with this handle for more information. • **PrimaryThread** is the thread on which NDIS initiated the operation. Usually, this is the first place to look, although the thread may have gone elsewhere if the operation is being handled asynchronously.	• 0x12 : The NET_PNP_EVENT_CODE of the stuck event. For possible values, see the previous list of values for item 2 in this list. • 0x13 : The NDIS_STATUS code of the stuck indication. Use **!ndiskd.help** to decode it. • 0x14 : 0 • 0x21 : 0 • 0x22 : 0 • 0x23 : The OID code of the stuck request. Use **!ndiskd.help** to decode it. • 0x24 : The OID code of the stuck request. Use **!ndiskd.help** to decode it. • 0x25 : 0 • 0x26 : 0

PARAMETER 1	PARAMETER 1 VALUE AND CAUSE OF ERROR	PARAMETER 2	PARAMETER 3	PARAMETER 4
		OTIFICATION to a filter driver. • 0x13 : NDIS_BUGCHECK_WATCHDOG_FILTER_STATUS_INDICATION There was a timeout while delivering a status indication to a filter driver. • 0x14 : NDIS_BUGCHECK_WATCHDOG_FILTER_DETACH There was a timeout while detaching a filter driver. • 0x21 : NDIS_BUGCHECK_WATCHDOG_MINIPORT_PAUSE There was a timeout while pausing a miniport adapter. • 0x22 : NDIS_BUGCHECK_WATCHDOG_MINIPORT_HALT There was a timeout while halting a miniport adapter. • 0x23 : NDIS_BUGCHECK_WATCHDOG_MINIPOR		

PARAMETER 1	PARAMETER 1 VALUE AND CAUSE OF ERROR	PARAMETER 2	PARAMETER 3	PARAMETER 4
		T_OID There was a timeout while delivering an OID request to a miniport adapter. • 0x24 : NDIS_BUGCHECK_WATCHDOG_FILTER_OID There was a timeout while delivering an OID request to a filter driver. • 0x25 : NDIS_BUGCHECK_WATCHDOG_MINIPORT_IDLE There was a timeout while idling a miniport adapter. • 0x26 : NDIS_BUGCHECK_WATCHDOG_CANCEL_IDLE There was a timeout while canceling an idle request on a miniport adapter.		

PARAMETER 1	PARAMETER 1 VALUE AND CAUSE OF ERROR	PARAMETER 2	PARAMETER 3	PARAMETER 4
0x26	NDIS_BUGCHECK_INVALID_OID_COMPLETION A miniport driver attempted to complete an OID request that is not currently pending on that miniport driver. This can be caused by the driver trying to complete the same request more than one time.	The miniport driver handle that caused the bugcheck. Run **!ndiskd.minidriver** with this handle for more information.	The NDIS OID request the miniport driver was trying to complete. You can try to run **!ndiskd.oid** with this request but the memory might not be valid at this point.	0
0x27	NDIS_BUGCHECK_LEAKED_NBL A driver has leaked a **NET_BUFFER_LIST** structure. Check with **!ndiskd.pendingnbls** to see any NBLs that are still pending on this driver.	Where the leak was detected. Possible values: • 0x01 : The leak was detected by the NBL tracker. The driver that is currently deregistering or unbinding is the most likely cause. Look at the callstack of the bugchecking thread. Drivers must not unbind or deregister while they still hold active NBLs.	0	0

Cause

Parameter 1 indicates the specific cause of the BUGCODE_NDIS_DRIVER bug check.

Remarks

The BUGCODE_NDIS_DRIVER bugcheck indendifies problems in network drivers. Often, the problem is caused by a NDIS miniport driver. You can get a complete list of NDIS miniport drivers by using **!ndiskd.netadapter**. You can get a bigger picture overview of the network stack with **!ndiskd.netreport**.

This bug check code occurs only on Microsoft Windows Server 2003 and later versions of Windows. In Windows 2000 and Windows XP, the corresponding code is **bug check 0xD2** (BUGCODE_ID_DRIVER).

Bug Check 0x7D: INSTALL_MORE_MEMORY

7/2/2019 • 2 minutes to read • Edit Online

The INSTALL_MORE_MEMORY bug check has a value of 0x0000007D. This bug check indicates that there is not enough memory to start up the Microsoft Windows operating system.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

INSTALL_MORE_MEMORY Parameters

PARAMETER	DESCRIPTION
1	The number of physical pages that are found
2	The lowest physical page
3	The highest physical page
4	0

Cause

The Windows operating system does not have sufficient memory to complete the startup process.

Resolution

Install more memory.

Bug Check 0x7E: SYSTEM_THREAD_EXCEPTION_NOT_HANDLED

10/15/2019 • 2 minutes to read • Edit Online

The SYSTEM_THREAD_EXCEPTION_NOT_HANDLED bug check has a value of 0x0000007E. This bug check indicates that a system thread generated an exception that the error handler did not catch.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

SYSTEM_THREAD_EXCEPTION_NOT_HANDLED parameters

PARAMETER	DESCRIPTION
1	The exception code that was not handled.
2	The address where the exception occurred.
3	The address of the exception record.
4	The address of the context record.

Cause

This bug check indicates that a system thread generated an exception that the error handler did not catch. To interpret it, you must identify which exception was generated.

Common exception codes include the following:

- 0x80000002: STATUS_DATATYPE_MISALIGNMENT indicates an unaligned data reference was encountered.
- 0x80000003: STATUS_BREAKPOINT indicates a breakpoint or ASSERT was encountered when no kernel debugger was attached to the system.
- 0xC0000005: STATUS_ACCESS_VIOLATION indicates a memory access violation occurred.

For a complete list of exception codes, see NTSTATUS values. The exception codes are defined in *ntstatus.h*, a header file provided by the Windows Driver Kit. (For more info, see Header files in the Windows Driver Kit).

Resolution

If you plan to debug this problem, the exception address (parameter 2) should identify the driver or function that caused this problem.

If a driver is listed by name within the bug check message, disable or remove that driver. If the issue is narrowed down to a single driver, set breakpoints and single-step forward in code to locate the failure and gain insight into events leading up to the crash.

The **!analyze** debugger extension displays information about the bug check and can be helpful in determining the root cause.

Additional analysis can be done by using the **!thread** extension, as well as the **dds**, **dps**, and **dqs** (display words and symbols) commands. This can be a reasonable technique when WinDbg reports "Probably caused by : ntkrnlmp.exe."

If exception code 0x80000003 occurs, a hard-coded breakpoint or assertion was hit, but the system was started with the **/NODEBUG** switch. This problem should not occur frequently. If it occurs repeatedly, make sure that a kernel debugger is connected and the system is started with the **/DEBUG** switch.

If exception code 0x80000002 occurs, the trap frame supplies additional information.

For more information about WinDbg and **!analyze**, see the following topics:

- Analyze crash dump files by using WinDbg
- Analyzing a kernel-mode dump file with WinDbg
- Using the !analyze extension and !analyze

Remarks

If you are not equipped to use the Windows debugger to work on this problem, you should use some basic troubleshooting techniques:

- Check the System Log in Event Viewer for additional error messages that might help identify the device or driver that is causing bug check 0x7E.
- If a driver is identified in the bug check message, disable the driver or check with the manufacturer for driver updates.
- Check with your hardware vendor for any ACPI or other firmware updates. Hardware issues, such as system incompatibilities, memory conflicts, and IRQ conflicts can also generate this error.
- You can also disable memory caching/shadowing of the BIOS to try to resolve the error. Also run hardware diagnostics that the system manufacturer supplies.
- Confirm that any new hardware that is installed is compatible with the installed version of Windows. For example, you can get information about required hardware at Windows 10 Specifications.

For additional general troubleshooting information, see Blue screen data.

Bug Check 0x7F: UNEXPECTED_KERNEL_MODE_TRAP

7/2/2019 • 5 minutes to read • Edit Online

The UNEXPECTED_KERNEL_MODE_TRAP bug check has a value of 0x0000007F. This bug check indicates that the Intel CPU generated a trap and the kernel failed to catch this trap.

This trap could be a *bound trap* (a trap the kernel is not permitted to catch) or a *double fault* (a fault that occurred while processing an earlier fault, which always results in a system failure).

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

UNEXPECTED_KERNEL_MODE_TRAP Parameters

The first parameter that appears on the blue screen specifies the trap number.

The most common trap codes include the following:

- 0x00000000, or Divide by Zero Error, indicates that a DIV instruction is executed and the divisor is zero. Memory corruption, other hardware problems, or software failures can cause this error.
- 0x00000004, or Overflow, occurs when the processor executes a call to an interrupt handler when the overflow (OF) flag is set.
- 0x00000005, or Bounds Check Fault, indicates that the processor, while executing a BOUND instruction, finds that the operand exceeds the specified limits. A BOUND instruction ensures that a signed array index is within a certain range.
- 0x00000006, or Invalid Opcode, indicates that the processor tries to execute an invalid instruction. This error typically occurs when the instruction pointer has become corrupted and is pointing to the wrong location. The most common cause of this error is hardware memory corruption.
- 0x00000008, or Double Fault, indicates that an exception occurs during a call to the handler for a prior exception. Typically, the two exceptions are handled serially. However, there are several exceptions that cannot be handled serially, and in this situation the processor signals a double fault. There are two common causes of a double fault:
  - A kernel stack overflow. This overflow occurs when a guard page is hit, and the kernel tries to push a trap frame. Because there is no stack left, a stack overflow results, causing the double fault. If you think this overview has occurred, use **!thread** to determine the stack limits, and then use **kb (Display Stack Backtrace)** with a large parameter (for example, **kb 100**) to display the full stack.
  - A hardware problem.

The less-common trap codes include the following:

- 0x00000001 -- A system-debugger call
- 0x00000003 -- A debugger breakpoint

- 0x00000007 -- A hardware coprocessor instruction with no coprocessor present
- 0x0000000A -- A corrupted Task State Segment
- 0x0000000B -- An access to a memory segment that was not present
- 0x0000000C -- An access to memory beyond the limits of a stack
- 0x0000000D -- An exception not covered by some other exception; a protection fault that pertains to access violations for applications

For other trap numbers, see an Intel architecture manual.

Cause

Bug check 0x7F typically occurs after you install a faulty or mismatched hardware (especially memory) or if installed hardware fails.

A double fault can occur when the kernel stack overflows. This overflow occurs if multiple drivers are attached to the same stack. For example, if two file system filter drivers are attached to the same stack and then the file system recurses back in, the stack overflows.

Resolution

Debugging: Always begin with the **!analyze** extension.

If this extension is not sufficient, use the **kv (Display Stack Backtrace)** debugger command.

- If **kv** shows a **taskGate**, use the **.tss (Display Task State Segment)** command on the part before the colon.
- If **kv** shows a trap frame, use the **.trap (Display Trap Frame)** command to format the frame.
- Otherwise, use the **.trap (Display Trap Frame)** command on the appropriate frame. (On x86-based platforms, this frame is associated with the procedure **NT!KiTrap**.)

After using one of these commands, use **kv** again to display the new stack.

Troubleshooting: If you recently added hardware to the computer, remove it to see if the error recurs. If existing hardware has failed, remove or replace the faulty component. Run hardware diagnostics that the system manufacturer supplies to determine which hardware component failed.

The memory scanner is especially important. Faulty or mismatched memory can cause this bug check. For more information about these procedures, see the owner's manual for your computer. Check that all adapter cards in the computer are properly seated. Use an ink eraser or an electrical contact treatment, available at electronics supply stores, to ensure adapter card contacts are clean.

If the error appears on a newly installed system, check the availability of updates for the BIOS, the SCSI controller, or network cards. These kind of updates are typically available on the Web site or BBS of the hardware manufacturer.

Confirm that all hard disk drives, hard disk controllers, and SCSI adapters are compatible with the installed version of Windows. For example, you can get information about compatibility with Windows 7 at the Windows 7 Compatibility Center.

If the error occurred after the installation of a new or updated device driver, you should remove or replace the driver. If, under this circumstance, the error occurs during the startup sequence and the system partition is formatted with NTFS, you might be able to use Safe Mode to rename or delete the faulty driver. If the driver is used as part of the system startup process in Safe Mode, you have to start the computer by using the Recovery

Console in order to access the file.

Also restart your computer, and then press F8 at the character-based menu that displays the operating system choices. At the **Advanced Options** menu, select the **Last Known Good Configuration** option. This option is most effective when you add only one driver or service at a time.

Overclocking (setting the CPU to run at speeds above the rated specification) can cause this error. If you have overclocked the computer that is experiencing the error, return the CPU to the default clock speed setting.

Check the System Log in Event Viewer for additional error messages that might help identify the device or driver that is causing the error. You can also disable memory caching of the BIOS to try to resolve the problem.

If you encountered this error while upgrading to a new version of the Windows operating system, the error might be caused by a device driver, a system service, a virus scanner, or a backup tool that is incompatible with the new version. If possible, remove all third-party device drivers and system services and disable any virus scanners before you upgrade. Contact the software manufacturer to obtain updates of these tools. Also make sure that you have installed the latest Windows Service Pack.

Finally, if all the above steps do not resolve the error, take the system motherboard to a repair facility for diagnostic testing. A crack, a scratched trace, or a defective component on the motherboard can also cause this error.

Bug Check 0x80: NMI_HARDWARE_FAILURE

7/2/2019 • 2 minutes to read • Edit Online

The NMI_HARDWARE_FAILURE bug check has a value of 0x00000080. This bug check indicates that a hardware malfunction has occurred.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

NMI_HARDWARE_FAILURE Parameters

None

Cause

A variety of hardware malfunctions can cause the NMI_HARDWARE_FAILURE bug check. The exact cause is difficult to determine.

Resolution

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause. Remove any hardware or drivers that have been recently installed. Make sure that all memory modules are of the same type.

Bug Check 0x81: SPIN_LOCK_INIT_FAILURE

7/2/2019 • 2 minutes to read • Edit Online

The SPIN_LOCK_INIT_FAILURE bug check has a value of 0x00000081.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Bug Check 0x82: DFS_FILE_SYSTEM

7/2/2019 • 2 minutes to read • Edit Online

The DFS_FILE_SYSTEM bug check has a value of 0x00000082.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Bug Check 0x85: SETUP_FAILURE

7/2/2019 • 2 minutes to read • Edit Online

The SETUP_FAILURE bug check has a value of 0x00000085. This bug check indicates that a fatal error occurred during setup.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

SETUP_FAILURE Parameters

Parameter 1 indicates the type of violation. Parameter 4 is not used. The meaning of the other parameters depends on the value of Parameter 1.

PARAMETER 1	PARAMETER 2	PARAMETER 3	CAUSE
0x0	0	0	The OEM HAL font is not a valid .fon format file, so setup cannot display text. This cause indicates that Vga*xxx*.fon on the boot floppy or CD is damaged.

PARAMETER 1	PARAMETER 2	PARAMETER 3	CAUSE
0x1	The precise video initialization failure: **0:NtCreateFile** of \device\video0 **1:** IOCTL_VIDEO_QUERY_NUM_AVAIL_MODES **2:** IOCTL_VIDEO_QUERY_AVAIL_MODES **3:** The desired video mode is not supported. This value indicates an internal setup error. **4:** IOCTL_VIDEO_SET_CURRENT_MODE (unable to set video mode) **5:** IOCTL_VIDEO_MAP_VIDEO_MEMORY **6:** IOCTL_VIDEO_LOAD_AND_SET_FONT	The status code from the NT API call, if appropriate	Video initialization failed. This failure might indicate that the disk that contains Vga.sys (or another video driver that is appropriate to the computer) is damaged or that the computer has video hardware that the Microsoft Windows operating system cannot communicate with.
0x2	0	0	Out of memory.
0x3	The precise keyboard initialization failure: **0:NtCreateFile** of \device\KeyboardClass0 failed. (Setup did not find a keyboard connected to the computer.) **1:** Unable to load keyboard layout DLL. (Setup could not load the keyboard layout file. This failure indicates that the CD or floppy disk is missing a file, such as Kbdus.dll for the U.S. release or another layout DLL for localized releases.)	0	Keyboard initialization failed. This failure might indicate that the disk that contains the keyboard driver (I8042prt.sys or Kbdclass.sys) is damaged or that the computer has keyboard hardware that Windows cannot communicate with. This failure might also mean that the keyboard layout DLL could not be loaded.

PARAMETER 1	PARAMETER 2	PARAMETER 3	CAUSE
0x4	0	0	Setup could not resolve the ARC device path name of the device that setup was started from. This error is an internal setup error.
0x5	Reserved	Reserved	Partitioning sanity check failed. This error indicates a bug in a disk driver.

Bug Check 0x8B: MBR_CHECKSUM_MISMATCH

7/2/2019 • 2 minutes to read • Edit Online

The MBR_CHECKSUM_MISMATCH bug check has a value of 0x0000008B. This bug check indicates that a mismatch has occurred in the MBR checksum.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

MBR_CHECKSUM_MISMATCH Parameters

PARAMETER	DESCRIPTION
1	The disk signature from MBR
2	The MBR checksum that the OS Loader calculates
3	The MBR checksum that the system calculates
4	Reserved

Cause

The MBR_CHECKSUM_MISMATCH bug check occurs during the boot process when the MBR checksum that the Microsoft Windows operating system calculates does not match the checksum that the loader passes in.

This error typically indicates a virus.

Resolution

There are many forms of viruses and not all can be detected. Typically, the newer viruses usually can be detected only by a virus scanner that has recently been upgraded. You should boot with a write-protected disk that contains a virus scanner and try to clean out the infection.

Bug Check 0x8E: KERNEL_MODE_EXCEPTION_NOT_HANDLED

7/2/2019 • 3 minutes to read • Edit Online

The KERNEL_MODE_EXCEPTION_NOT_HANDLED bug check has a value of 0x0000008E. This bug check indicates that a kernel-mode application generated an exception that the error handler did not catch.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

KERNEL_MODE_EXCEPTION_NOT_HANDLED Parameters

PARAMETER	DESCRIPTION
1	The exception code that was not handled
2	The address where the exception occurred
3	The trap frame
4	Reserved

Cause

The KERNEL_MODE_EXCEPTION_NOT_HANDLED bug check is a very common bug check. To interpret it, you must identify which exception was generated.

Common exception codes include the following:

- 0x80000002: STATUS_DATATYPE_MISALIGNMENT indicates that an unaligned data reference was encountered.
- 0x80000003: STATUS_BREAKPOINT indicates that a breakpoint or ASSERT was encountered when no kernel debugger was attached to the system.
- 0xC0000005: STATUS_ACCESS_VIOLATION indicates that a memory access violation occurred.

For a complete list of exception codes, see the Ntstatus.h file that is located in the inc directory of the Microsoft Windows Driver Kit (WDK).

Resolution

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause. If you are not equipped to debug this problem, you should use some basic troubleshooting techniques:

- Make sure you have enough disk space.
- If a driver is identified in the bug check message, disable the driver or check with the manufacturer for driver updates.
- Try changing video adapters.
- Check with your hardware vendor for any BIOS updates.
- Disable BIOS memory options such as caching or shadowing.

If you plan to debug this problem, you might find it difficult to obtain a stack trace. Parameter 2 (the exception address) should identify the driver or function that caused this problem.

If exception code 0x80000003 occurs, a hard-coded breakpoint or assertion was hit, but the computer was started with the **/NODEBUG** switch. This problem should rarely occur. If it occurs repeatedly, make sure that a kernel debugger is connected and that the computer is started with the **/DEBUG** switch.

If exception code 0x80000002 occurs, the trap frame supplies additional information.

If you do not know the specific cause of the exception, consider the following items:

- Hardware incompatibility. Make sure that any new hardware that is installed is compatible with the installed version of Windows. For example, you can get information about compatibility with Windows 7 at the Windows 7 Compatibility Center.
- Faulty device driver or system service. A faulty device driver or system service might be responsible for this error. Hardware issues, such as BIOS incompatibilities, memory conflicts, and IRQ conflicts can also generate this error.

If the bug check message lists a driver by name , disable or remove that driver. Also, disable or remove any drivers or services that were recently added. If the error occurs during the startup sequence and the system partition is formatted with NTFS file system, you might be able to use Safe Mode to rename or delete the faulty driver. If the driver is used as part of the system startup process in Safe Mode, you have to start the computer by using the Recovery Console to access the file.

If the problem is associated with Win32k.sys, the source of the error might be a third-party remote control program. If such software is installed, you can remove the service by starting the system by using the Recovery Console and then deleting the offending system service file.

Check the System Log in Event Viewer for additional error messages that might help identify the device or driver that is causing bug check 0x8E. You can disable memory caching of the BIOS to try to resolve the error. You should also run hardware diagnostics, especially the memory scanner, that the system manufacturer supplies. For more information about these procedures, see the owner's manual for your computer.

The error that generates this message can occur after the first restart during Windows Setup, or after Setup is finished. A possible cause of the error is lack of disk space for installation and system BIOS incompatibilities. For problems during Windows installation that are associated with lack of disk space, reduce the number of files on the target hard disk drive. Check for and delete any temporary files that you do not have to have, Internet cache files, application backup files, and .chk files that contain saved file fragments from disk scans. You can also use another hard disk drive with more free space for the installation.

You can resolve BIOS problems by upgrading the system BIOS version.

Bug Check 0x8F: PP0_INITIALIZATION_FAILED

7/2/2019 • 2 minutes to read • Edit Online

The PP0_INITIALIZATION_FAILED bug check has a value of 0x0000008F. This bug check indicates that the Plug and Play (PnP) manager could not be initialized.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

PP0_INITIALIZATION_FAILED Parameters

None

Cause

An error occurred during Phase 0 initialization of the kernel-mode PnP manager.

Bug Check 0x90: PP1_INITIALIZATION_FAILED

7/2/2019 • 2 minutes to read • Edit Online

The PP1_INITIALIZATION_FAILED bug check has a value of 0x00000090. This bug check indicates that the Plug and Play (PnP) manager could not be initialized.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

PP1_INITIALIZATION_FAILED Parameters

None

Cause

An error occurred during Phase 1 initialization of the kernel-mode PnP manager.

Phase 1 is where most of the initialization is done, including setting up the registry files and other environment settings for drivers to call during the subsequent I/O initialization.

Bug Check 0x92: UP_DRIVER_ON_MP_SYSTEM

7/2/2019 • 2 minutes to read • Edit Online

The UP_DRIVER_ON_MP_SYSTEM bug check has a value of 0x00000092. This bug check indicates that a uniprocessor-only driver has been loaded on a multiprocessor system.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

UP_DRIVER_ON_MP_SYSTEM Parameters

PARAMETER	DESCRIPTION
1	The base address of the driver
2	Reserved
3	Reserved
4	Reserved

Cause

A driver that is compiled to work only on uniprocessor machines has been loaded, but the Microsoft Windows operating system is running on a multiprocessor system with more than one active processor.

Resolution

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause.

Bug Check 0x93: INVALID_KERNEL_HANDLE

10/10/2019 • 2 minutes to read • Edit Online

The INVALID_KERNEL_HANDLE bug check has a value of 0x00000093. This bug check indicates that an invalid or protected handle was passed to **NtClose**.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

INVALID_KERNEL_HANDLE Parameters

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
The handle that NtClose was called with	0	0	0	A protected handle was closed.
The handle that NtClose was called with	1	0	0	An invalid handle was closed or referenced.
The handle that was referenced	The handle table	0	1	The error occurred referencing an invalid kernel handle and bad handle detection was enabled.

Cause

The INVALID_KERNEL_HANDLE bug check indicates that some kernel code (for example, a server, redirector, or another driver) tried to close an invalid handle or a protected handle.

If parameter 4 has a value of 1, this indicates that the error occurred referencing an invalid kernel handle and bad handle detection was enabled.

This message occurs if kernel code attempts to close or reference a handle that is not a valid handle. Only invalid or protected handles passed to NtClose will cause this bugcheck, unless bad handle detection is enabled.

Bug Check 0x94: KERNEL_STACK_LOCKED_AT_EXIT

7/2/2019 • 2 minutes to read • Edit Online

The KERNEL_STACK_LOCKED_AT_EXIT bug check has a value of 0x00000094. This bug check indicates that a thread exited while its kernel stack was marked as not swappable

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

KERNEL_STACK_LOCKED_AT_EXIT Parameters

None

Bug Check 0x96: INVALID_WORK_QUEUE_ITEM

7/2/2019 • 2 minutes to read • Edit Online

The INVALID_WORK_QUEUE_ITEM bug check has a value of 0x00000096. This bug check indicates that a queue entry was removed that contained a **NULL** pointer.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

INVALID_WORK_QUEUE_ITEM Parameters

PARAMETER	DESCRIPTION
1	The address of the queue entry whose **flink** or **blink** field is **NULL**.
2	The address of the queue that is being referenced. Typically, this queue is an **ExWorkerQueue**.
3	The base address of the **ExWorkerQueue** array. (This address helps you determine if the queue in question is indeed an **ExWorkerQueue**. If the queue is an **ExWorkerQueue**, the offset from this parameter will isolate the queue.)
4	Assuming the queue is an **ExWorkerQueue**, this value is the address of the worker routine that would have been called if the work item had been valid. (You can use this address to isolate the driver that is misusing the work queue.)

Cause

The INVALID_WORK_QUEUE_ITEM bug check occurs when **KeRemoveQueue** removes a queue entry whose **flink** or **blink** field is **NULL**.

Any queue misuse can cause this error. But typically this error occurs because worker thread work items are misused.

An entry on a queue can be inserted on the list only one time. When an item is removed from a queue, its **flink** field is set to **NULL**. Then, when this item is removed the second time, this bug check occurs.

In most situations, the queue that is being referenced is an **ExWorkerQueue** (executive worker queue). To help identify the driver that caused the error, Parameter 4 displays the address of the worker routine that would have been called if this work item had been valid. However, if the queue that is being referenced is not an **ExWorkerQueue**, this parameter is not useful.

Bug Check 0x97: BOUND_IMAGE_UNSUPPORTED

7/2/2019 • 2 minutes to read • Edit Online

The BOUND_IMAGE_UNSUPPORTED bug check has a value of 0x00000097.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Bug Check 0x98: END_OF_NT_EVALUATION_PERIOD

7/2/2019 • 2 minutes to read • Edit Online

The END_OF_NT_EVALUATION_PERIOD bug check has a value of 0x00000098. This bug check indicates that the trial period for the Microsoft Windows operating system has ended.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

END_OF_NT_EVALUATION_PERIOD Parameters

PARAMETER	DESCRIPTION
1	The low-order 32 bits of the product expiration date
2	The high-order 32 bits of the product expiration date
3	Reserved
4	Reserved

Cause

Your installation of the Windows operating system is an evaluation unit with an expiration date. The trial period is over.

Bug Check 0x99: INVALID_REGION_OR_SEGMENT

7/2/2019 • 2 minutes to read • Edit Online

The INVALID_REGION_OR_SEGMENT bug check has a value of 0x00000099. This bug check indicates that **ExInitializeRegion** or **ExInterlockedExtendRegion** was called with an invalid set of parameters.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

INVALID_REGION_OR_SEGMENT Parameters

None

Bug Check 0x9A: SYSTEM_LICENSE_VIOLATION

7/2/2019 • 3 minutes to read • Edit Online

The SYSTEM_LICENSE_VIOLATION bug check has a value of 0x0000009A. This bug check indicates that the software license agreement has been violated.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

SYSTEM_LICENSE_VIOLATION Parameters

Parameter 1 indicates the type of violation. The meaning of the other parameters depends on the value of Parameter 1.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE
0x00	**0:** The product should be WinNT **1:** The product should be LanmanNT or ServerNT	A partial serial number	The first two characters of the product type from the product options	Offline product type changes have been attempted.
0x01	The registered evaluation time from source 1	A partial serial number	The registered evaluation time from an alternate source	Offline changes to the Microsoft Windows evaluation unit time period have been attempted.
0x02	The status code that is associated with the open failure	0	0	The setup key could not be opened.
0x03	The status code that is associated with the key lookup failure	0	0	The SetupType or SetupInProgress value from the setup key is missing, so setup mode could not be detected.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE
0x04	The status code that is associated with the key lookup failure	0	0	The **SystemPrefix** value from the setup key is missing.
0x05	(See the setup code)	An invalid value was found in licensed processors	The officially licensed number of processors	Offline changes to the number of licensed processors have been attempted.
0x06	The status code that is associated with the open failure	0	0	The **ProductOptions** key could not be opened.
0x07	The status code that is associated with the read failure	0	0	The **ProductType** value could not be read.
0x08	The status code that is associated with the Change Notify failure	0	0	Change Notify on **ProductOptions** failed.
0x09	The status code that is associated with the Change Notify failure	0	0	Change Notify on **SystemPrefix** failed.
0x0A	0	0	0	An NTW system was converted to an NTS system.
0x0B	The status code that is associated with the change failure	0	0	The reference of the setup key failed.
0x0C	The status code that is associated with the change failure	0	0	The reference of the product options key failed.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE
0x0D	The status code that is associated with the failure	0	0	The attempt to open **ProductOptions** in the worker thread failed.
0x0F	The status code that is associated with the failure	0	0	The attempt to open the setup key failed.
0x10	The status code that is associated with the failure	**0:** set value failed **1:** Change Notify failed	0	A failure occurred in the setup key worker thread.
0x11	The status code that is associated with the failure	**0:** set value failed **1:** Change Notify failed	0	A failure occurred in the product options key worker thread.
0x12	The status code that is associated with the failure	0	0	Unable to open the **LicenseInfoSuites** key for the suite.
0x13	The status code that is associated with the failure	0	0	Unable to query the **LicenseInfoSuites** key for the suite.
0x14	The size of the memory allocation	0	0	Unable to allocate memory.
0x15	The status code that is associated with the failure	Reserved	0	Unable to reset the **ConcurrentLimit** value for the suite key.
0x16	The status code that is associated with the failure	0	0	Unable to open the license key for a suite product.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE
0x17	The status code that is associated with the failure	0	0	Unable to reset the **ConcurrentLimit** value for a suite product.
0x18	The status code that is associated with the open failure	Reserved	0	Unable to start the Change Notify for the **LicenseInfoSuites**.
0x19	0	0	0	A suite is running on a system that must be PDC.
0x1A	The status code that is associated with the failure	0	0	A failure occurred when enumerating the suites.
0x1B	0	0	0	Changes to the policy cache were attempted.

Cause

The Microsoft Windows operating system detects a violation of the software license agreement.

A user might have tried to change the product type of an offline system or change the trial period of an evaluation unit of Windows. For more information about the specific violation, see the parameter list.

Bug Check 0x9B: UDFS_FILE_SYSTEM

7/2/2019 • 2 minutes to read • Edit Online

The UDFS_FILE_SYSTEM bug check has a value of 0x0000009B. This bug check indicates that a problem occurred in the UDF file system.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

UDFS_FILE_SYSTEM Parameters

PARAMETER	DESCRIPTION
1	The source file and line number information. The high 16 bits (the first four hexadecimal digits after the "0x") identify the source file by its identifier number. The low 16 bits identify the source line in the file where the bug check occurred.
2	If **UdfExceptionFilter** is on the stack, this parameter specifies the address of the exception record.
3	If **UdfExceptionFilter** is on the stack, this parameter specifies the address of the context record.
4	Reserved.

Cause

The UDFS_FILE_SYSTEM bug check might be caused disk corruption. Corruption in the file system or bad blocks (sectors) on the disk can induce this error. Corrupted SCSI and IDE drivers can also adversely affect the system's ability to read and write to the disk and cause the error.

This bug check might also occur if nonpaged pool memory is full. If the nonpaged pool memory is full, this error can stop the system. However, during the indexing process, if the amount of available nonpaged pool memory is very low, another kernel-mode driver that requires nonpaged pool memory can also trigger this error.

Resolution

To debug this problem: Use the **.cxr (Display Context Record)** command with Parameter 3, and then use **kb (Display Stack Backtrace)**.

To resolve a disk corruption problem: Check Event Viewer for error messages from SCSI and FASTFAT (System Log) or Autochk (Application Log) that might help identify the device or driver that is causing the error. Disable any virus scanners, backup application, or disk defragmenter tools that continually monitor the system. You should also run hardware diagnostics that the system manufacturer supplies. For more information about these

procedures, see the owner's manual for your computer. Run **Chkdsk /f /r** to detect and resolve any file system structural corruption. You must restart the system before the disk scan begins on a system partition.

To resolve a nonpaged pool memory depletion problem: Add new physical memory to the computer. This memory increases the quantity of nonpaged pool memory that is available to the kernel.

Bug Check 0x9C: MACHINE_CHECK_EXCEPTION

7/2/2019 • 2 minutes to read • Edit Online

The MACHINE_CHECK_EXCEPTION bug check has a value of 0x0000009C. This bug check indicates that a fatal machine check exception has occurred.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

MACHINE_CHECK_EXCEPTION Parameters

The four parameters that are listed in the message have different meanings, depending on the processor type.

If the processor is based on an older x86-based architecture and has the Machine Check Exception (MCE) feature but not the Machine Check Architecture (MCA) feature (for example, the Intel Pentium processor), the parameters have the following meaning.

PARAMETER	DESCRIPTION
1	The low 32 bits of P5_MC_TYPE Machine Service Report (MSR)
2	The address of the MCA_EXCEPTION structure
3	The high 32 bits of P5_MC_ADDR MSR
4	The low 32 bits of P5_MC_ADDR MSR

If the processor is based on a newer x86-based architecture and has the MCA feature and the MCE feature (for example, any Intel Processor of family 6 or higher, such as Pentium Pro, Pentium IV, or Xeon), or if the processor is an x64-based processor, the parameters have the following meaning.

PARAMETER	DESCRIPTION
1	The bank number
2	The address of the MCA_EXCEPTION structure
3	The high 32 bits of MCi_STATUS MSR for the MCA bank that had the error

PARAMETER	DESCRIPTION
4	The low 32 bits of MCi_STATUS MSR for the MCA bank that had the error

On an Itanium-based processor, the parameters have the following meaning.

Note Parameter 1 indicates the type of violation.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE
0x1	The address of the log	The size of the log	0	
0x2	The address of the log	The size of the log	The error code	The system abstraction layer (SAL) returned an error for SAL_GET_STATEINFO while processing MCA.
0x3	The address of the log	The size of the log	The error code	SAL returned an error for SAL_CLEAR_STATEINFO while it processed MCA.
0x4	The address of the log	The size of the log	0	Firmware (FW) reported a fatal MCA.
0x5	The address of the log	The size of the log	0	There are two possible causes: • SAL reported a recoverable MCA, but this recovery is not currently supported. • SAL generated an MCA but could not produce an error record.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE
0xB	The address of the log	The size of the log	0	
0xC	The address of the log	The size of the log	The error code	SAL returned an error for SAL_GET_STATEINFO while processing an INIT event.
0xD	The address of the log	The size of the log	The error code	SAL returned an error for SAL_CLEAR_STATEINFO while it processed an INIT event.
0xE	The address of the log	The size of the log	0	

Remarks

In Windows Vista and later operating systems, this bug check occurs only in the following circumstances.

- WHEA is not fully initialized.
- All processors that rendezvous have no errors in their registers.

For other circumstances, this bug check has been replaced with **bug Check 0x124: WHEA_UNCORRECTABLE_ERROR** in Windows Vista and later operating systems.

For more information about Machine Check Architecture (MCA), see the Intel or AMD Web sites.

Bug Check 0x9E: USER_MODE_HEALTH_MONITOR

7/2/2019 • 2 minutes to read • Edit Online

The USER_MODE_HEALTH_MONITOR bug check has a value of 0x0000009E. This bug check indicates that one or more critical user-mode components failed to satisfy a health check.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

USER_MODE_HEALTH_MONITOR Parameters

PARAMETER	DESCRIPTION
1	The process that failed to satisfy a health check in the configured time-out
2	The health monitoring time-out, in seconds
3	Watchdog source. In combination with process address helps to identify what sub-component has created this watchdog. Values listed below.
4	Reserved

VALUES

```
  0  : WatchdogSourceDefault
       Source was not specified
  1  : WatchdogSourceRhsCleanup
       Monitors that RHS process goes away when
       terminating on graceful exit
  2  : WatchdogSourceRhsResourceDeadlockBugcheckNow
       RHS was asked to immediately bugcheck machine
       on resource deadlock
  3  : WatchdogSourceRhsExceptionFromResource
       Resource has leaked unhandled exception from an entry point,
       RHS is terminating and this watchdog monitors that
       process will go away
  4  : WatchdogSourceRhsUnhandledException
       Unhandled exception in RHS.
       RHS is terminating and this watchdog monitors that
       process will go away
  5  : WatchdogSourceRhsResourceDeadlock
       Monitors that RHS process goes away when
       terminating on resource deadlock
  6  : WatchdogSourceRhsResourceTypeDeadlock
       Monitors that RHS process goes away when
       terminating on resource type deadlock
  7  : WatchdogSourceClussvcUnhandledException
       Unhandled exception in clussvc.
       clussvc is terminating and this watchdog monitors that
       process will go away
  8  : WatchdogSourceClussvcBugcheckMessageRecieved
       Another cluster node has sent message asking to bugcheck this node.
  9  : WatchdogSourceClussvcWatchdogBugcheck
       User mode watchdog has expired and created netft watchdog
       to bugchecked the node.
 0xA : WatchdogSourceClussvcIsAlive
       Cluster service sends heartbeat to netft every 500 millseconds.
       By default, netft expects at least 1 heartbeat per second.
       If this watchdog was triggered that means clussvc is not getting
       CPU to send heartbeats.
0x65 : WatchdogSourceRhsResourceDeadlockPhysicalDisk
        A subclass of WatchdogSourceRhsResourceDeadlock.
0x66 : WatchdogSourceRhsResourceDeadlockStoragePool
        A subclass of WatchdogSourceRhsResourceDeadlock.
0x67 : WatchdogSourceRhsResourceDeadlockFileServer
        A subclass of WatchdogSourceRhsResourceDeadlock.
0x68 : WatchdogSourceRhsResourceDeadlockSODAFileServer
        A subclass of WatchdogSourceRhsResourceDeadlock.
0x69 : WatchdogSourceRhsResourceDeadlockStorageReplica
        A subclass of WatchdogSourceRhsResourceDeadlock.
0x6A : WatchdogSourceRhsResourceDeadlockStorageQOS
        A subclass of WatchdogSourceRhsResourceDeadlock.
0x6B : WatchdogSourceRhsResourceDeadlockStorageNFSV2
        A subclass of WatchdogSourceRhsResourceDeadlock.
0xC9 : WatchdogSourceRhsResourceTypeDeadlockPhysicalDisk
        A subclass of WatchdogSourceRhsResourceTypeDeadlock.
0xCA : WatchdogSourceRhsResourceTypeDeadlockStoragePool
        A subclass of WatchdogSourceRhsResourceTypeDeadlock.
0xCB : WatchdogSourceRhsResourceTypeDeadlockFileServer
        A subclass of WatchdogSourceRhsResourceTypeDeadlock.
0xCC : WatchdogSourceRhsResourceTypeDeadlockSODAFileServer
        A subclass of WatchdogSourceRhsResourceTypeDeadlock.
0xCD : WatchdogSourceRhsResourceTypeDeadlockStorageReplica
        A subclass of WatchdogSourceRhsResourceTypeDeadlock.
0xCE : WatchdogSourceRhsResourceTypeDeadlockStorageQOS
        A subclass of WatchdogSourceRhsResourceTypeDeadlock.
0xCF : WatchdogSourceRhsResourceTypeDeadlockStorageNFSV2
        A subclass of WatchdogSourceRhsResourceTypeDeadlock.
```

Cause

Hardware mechanisms, such as watchdog timers, can detect that basic kernel services are not executing. However, resource starvation issues (including memory leaks, lock contention, and scheduling priority misconfiguration) can block critical user-mode components without blocking deferred procedure calls (DPCs) or draining the non-paged pool.

Kernel components can extend watchdog timer functionality to user mode by periodically monitoring critical applications. This bug check indicates that a user-mode health check failed in a way that prevents graceful shutdown. This bug check restores critical services by restarting or enabling application failover to other servers.

(Developer Content) Bug Check 0x9F: DRIVER_POWER_STATE_FAILURE

8/26/2019 • 7 minutes to read • Edit Online

The DRIVER_POWER_STATE_FAILURE bug check has a value of 0x0000009F. This bug check indicates that the driver is in an inconsistent or invalid power state.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

DRIVER_POWER_STATE_FAILURE Parameters

Parameter 1 indicates the type of violation.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE
0x1	The device object	Reserved	Reserved	The device object that is being freed still has an outstanding power request that it has not completed.
0x2	The target device's device object, if it is available	The device object	The driver object, if it is available	The device object completed the I/O request packet (IRP) for the system power state request, but it did not call **PoStartNextPowerIrp**.
0x3	The physical device object (PDO) of the stack	The functional device object (FDO) of the stack. In Windows 7 and later, nt!TRIAGE_9F_POWER.	The blocked IRP	A device object has been blocking an IRP for too long a time.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE
0x4	Time-out value, in seconds.	The thread currently holding onto the Plug-and-Play (PnP) lock.	In Windows 7 and later, nt!TRIAGE_9F_POWER.	The power state transition timed out waiting to synchronize with the PnP subsystem.
0x500	Reserved	The target device's device object, if available	Device object	The device object completed the IRP for the system power state request, but it did not call **PoStartNextPowerIrp**.

Cause

For a description of the possible causes, see the description of each code in the Parameters section.

Resolution

Debugging bug check 0x9F when Parameter 1 equals 0x3

- In a kernel debugger, use the **!analyze -v** command to perform the initial bug check analysis. The verbose analysis displays the address of the **nt!TRIAGE_9F_POWER** structure, which is in Arg3.

```
kd>!analyze -v
*******************************************************************************
*                                                                             *
*                        Bugcheck Analysis                                    *
*                                                                             *
*******************************************************************************

    DRIVER_POWER_STATE_FAILURE (9f)
    A driver has failed to complete a power IRP within a specific time.
    Arguments:
    Arg1: 0000000000000003, A device object has been blocking an Irp for too long a time
    Arg2: fffffa8007b13440, Physical Device Object of the stack
    Arg3: fffff8000386c3d8, nt!TRIAGE_9F_POWER on Win7 and higher, otherwise the Functional Device Object of
the stack
    Arg4: fffffa800ab61bd0, The blocked IRP
```

The nt!TRIAGE_9F_POWER structure provides additional bug check information that might help you determine the cause of this bug check. The structure can provide a list of all outstanding power IRPs, a list of all power IRP worker threads, and a pointer to the delayed system worker queue.

- Use the **dt (Display Type)** command and specify the nt!TRIAGE_9F_POWER structure using the address from Arg3.

```
0: kd> dt nt!TRIAGE_9F_POWER fffff8000386c3d8
   +0x000 Signature        : 0x8000
   +0x002 Revision         : 1
   +0x008 IrpList          : 0xfffff800`01c78bd0 _LIST_ENTRY [ 0xfffffa80`09f43620 - 0xfffffa80`0ad00170 ]
   +0x010 ThreadList       : 0xfffff800`01c78520 _LIST_ENTRY [ 0xfffff880`009cdb98 - 0xfffff880`181f2b98 ]
   +0x018 DelayedWorkQueue : 0xfffff800`01c6d2d8 _TRIAGE_EX_WORK_QUEUE
```

The **dt (Display Type)** command displays the structure. You can use various debugger commands to follow the LIST_ENTRY fields to examine the list of outstanding IRPs and the power IRP worker threads.

- Use the **!irp** command to examine the IRP that was blocked. The address of this IRP is in Arg4.

```
0: kd> !irp fffffa800ab61bd0
Irp is active with 7 stacks 6 is current (= 0xfffffa800ab61e08)
 No Mdl: No System Buffer: Thread 00000000:  Irp stack trace.
     cmd  flg cl Device   File     Completion-Context
 [N/A(0), N/A(0)]
            0  0 00000000 00000000 00000000-00000000

            Args: 00000000 00000000 00000000 00000000
 [N/A(0), N/A(0)]
            0  0 00000000 00000000 00000000-00000000

            Args: 00000000 00000000 00000000 00000000
 [N/A(0), N/A(0)]
            0  0 00000000 00000000 00000000-00000000

            Args: 00000000 00000000 00000000 00000000
 [N/A(0), N/A(0)]
            0  0 00000000 00000000 00000000-00000000

            Args: 00000000 00000000 00000000 00000000
 [N/A(0), N/A(0)]
            0  0 00000000 00000000 00000000-00000000

            Args: 00000000 00000000 00000000 00000000
>[IRP_MJ_POWER(16), IRP_MN_SET_POWER(2)]
            0 e1 fffffa800783f060 00000000 00000000-00000000    pending
           \Driver\HidUsb
            Args: 00016600 00000001 00000004 00000006
 [N/A(0), N/A(0)]
            0  0 00000000 00000000 00000000-fffffa800ad00170

            Args: 00000000 00000000 00000000 00000000
```

- Use the **!devstack** command with the PDO address in Arg2, to display information associated with the faulting driver.

```
0: kd> !devstack fffffa8007b13440
  !DevObj           !DrvObj            !DevExt           ObjectName
  fffffa800783f060  \Driver\HidUsb     fffffa800783f1b0  InfoMask field not found for _OBJECT_HEADER at
fffffa800783f030

> fffffa8007b13440  \Driver\usbhub     fffffa8007b13590  Cannot read info offset from
nt!ObpInfoMaskToOffset

!DevNode fffffa8007ac8a00 :
  DeviceInst is "USB\VID_04D8&PID_0033\5&46fa7b7&0&1"
  ServiceName is "HidUsb"
```

- Use the !poaction command to display the threads that handle the power operations and any allocated power IRPs.

```
3: kd> !poaction
PopAction: fffff801332f3fe0
  State.....: 0 - Idle
  Updates.....: 0
  Action.....: None
  Lightest State.: Unspecified
  Flags.....: 10000003 QueryApps|UIAllowed
  Irp minor.....: ??
  System State...: Unspecified
  Hiber Context..: 0000000000000000

Allocated power irps (PopIrpList - fffff801332f44f0)
  IRP: ffffe0001d53d8f0 (wait-wake/S0), PDO: ffffe00013cae060
  IRP: ffffe0001049a5d0 (wait-wake/S0), PDO: ffffe00012d42050
  IRP: ffffe00013d07420 (set/D3,), PDO: ffffe00012daf840, CURRENT: ffffe00012dd5040
  IRP: ffffe0001e5ac5d0 (wait-wake/S0), PDO: ffffe00013d33060
  IRP: ffffe0001ed3e420 (wait-wake/S0), PDO: ffffe00013c96060
  IRP: ffffe000195fe010 (wait-wake/S0), PDO: ffffe00012d32050

Irp worker threads (PopIrpThreadList - fffff801332f3100)
  THREAD: ffffe0000ef5d040 (static)
  THREAD: ffffe0000ef5e040 (static), IRP: ffffe00013d07420, DEVICE: ffffe00012dd5040

PopAction: fffff801332f3fe0
  State.....: 0 - Idle
  Updates.....: 0
  Action.....: None
  Lightest State.: Unspecified
  Flags.....: 10000003 QueryApps|UIAllowed
  Irp minor.....: ??
  System State...: Unspecified
  Hiber Context..: 0000000000000000

Allocated power irps (PopIrpList - fffff801332f44f0)
  IRP: ffffe0001d53d8f0 (wait-wake/S0), PDO: ffffe00013cae060
  IRP: ffffe0001049a5d0 (wait-wake/S0), PDO: ffffe00012d42050
  IRP: ffffe00013d07420 (set/D3,), PDO: ffffe00012daf840, CURRENT: ffffe00012dd5040
  IRP: ffffe0001e5ac5d0 (wait-wake/S0), PDO: ffffe00013d33060
  IRP: ffffe0001ed3e420 (wait-wake/S0), PDO: ffffe00013c96060
  IRP: ffffe000195fe010 (wait-wake/S0), PDO: ffffe00012d32050

Irp worker threads (PopIrpThreadList - fffff801332f3100)
  THREAD: ffffe0000ef5d040 (static)
  THREAD: ffffe0000ef5e040 (static), IRP: ffffe00013d07420, DEVICE: ffffe00012dd5040
```

- If you are working with a KMDF driver, use the Windows Driver Framework Extensions (!wdfkd) to gather additional information.

  Use **!wdfkd.wdflogdump** <your driver name>, to see if KMDF is waiting for you to ACK any pending requests.

  Use **!wdfkd.wdfdevicequeues** <your WDFDEVICE> to examine all outstanding requests and what state they are in.

- Use the **!stacks** extension to examine the state of every thread and look for a thread that might be holding up the power state transition.

- To help you determine the cause of the error, consider the following questions:

  - What are the characteristics of the physical device object (PDO) driver (Arg2)?
  - Can you find the blocked thread? When you examine the thread with the **!thread** debugger command, what does the thread consist of?
  - Is there IO associated with the thread that is blocking it? What symbols are on the stack?

- When you examine the blocked power IRP, what do you notice?
- What is the PnP minor function code of the power IRP?

Debugging bug check 0x9F when Parameter 1 equals 0x4

- In a kernel debugger, use the **!analyze -v** command to perform the initial bug check analysis. The verbose analysis displays the address of the **nt!TRIAGE_9F_PNP** structure, which is in Parameter 4 (arg4).

```
kd> !analyze -v
*******************************************************************************
*                                                                             *
*                        Bugcheck Analysis                                    *
*                                                                             *
*******************************************************************************

DRIVER_POWER_STATE_FAILURE (9f)
A driver has failed to complete a power IRP within a specific time (usually 10 minutes).
Arguments:
Arg1: 00000004, The power transition timed out waiting to synchronize with the Pnp
        subsystem.
Arg2: 00000258, Timeout in seconds.
Arg3: 84e01a70, The thread currently holding on to the Pnp lock.
Arg4: 82931b24, nt!TRIAGE_9F_PNP on Win7
```

The nt!TRIAGE_9F_PNP structure provides additional bug check information that might help you determine the cause of the error. The nt!TRIAGE_9F_PNP structure provides a pointer to a structure that contains the list of dispatched (but not completed) PnP IRPs and provides a pointer to the delayed system worker queue.

- Use the **dt (Display Type)** command and specify the **nt!TRIAGE_9F_PNP** structure and the address that you found in Arg4.

```
kd> dt nt!TRIAGE_9F_PNP 82931b24
   +0x000 Signature        : 0x8001
   +0x002 Revision         : 1
   +0x004 CompletionQueue  : 0x82970e20 _TRIAGE_PNP_DEVICE_COMPLETION_QUEUE
   +0x008 DelayedWorkQueue : 0x829455bc _TRIAGE_EX_WORK_QUEUE
```

The **dt (Display Type)** command displays the structure. You can use debugger commands to follow the LIST_ENTRY fields to examine the list of outstanding PnP IRPs.

To help you determine the cause of the error, consider the following questions:

- Is there an IRP associated with the thread?
- Is there any IO in the CompletionQueue?
- What symbols are on the stack?
- Refer to the additional techniques described above under parameter 0x3.

Remarks

If you are not equipped to debug this problem using the techniques described above, you can use some basic troubleshooting techniques.

- If new device drivers or system services have been added recently, try removing or updating them. Try to determine what changed in the system that caused the new bug check code to appear.

- Look in **Device Manager** to see if any devices are marked with the exclamation point (!). Review the events log displayed in driver properties for any faulting driver. Try updating the related driver.
- Check the System Log in Event Viewer for additional error messages that might help pinpoint the device or driver that is causing the error. For more information, see Open Event Viewer. Look for critical errors in the system log that occurred in the same time window as the blue screen.
- To try and isolate the cause, temporally disable power save using control panel, power options. Some driver issues are related to the various states of system hibernation and the suspending and resumption of power.
- If you recently added hardware to the system, try removing or replacing it. Or check with the manufacturer to see if any patches are available.
- You can try running the hardware diagnostics supplied by the system manufacturer.
- Check with the manufacturer to see if an updated system ACPI/BIOS or other firmware is available.

Bug Check 0xA0: INTERNAL_POWER_ERROR

7/2/2019 • 9 minutes to read • Edit Online

The INTERNAL_POWER_ERROR bug check has a value of 0x000000A0. This bug check indicates that the power policy manager experienced a fatal error.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

INTERNAL_POWER_ERROR Parameters

Parameter 1 indicates the type of violation. The meaning of the other parameters depends on the value of Parameter 1.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE
0x1	**1:** A device has overrun its maximum number of reference counts. **2, 3, or 4:** Too many inrush power IRPs have been queued. **5:** The power IRP has been sent to a passive level device object. **6:** The system has failed to allocate a necessary power IRP.	If Parameter 2 has a value of 1, the maximum number of references allowed. If Parameter 2 has a value of 2, 3, or 4, the maximum number of pending IRPs allowed. If Parameter 2 has a value of 6, the target device object.	If Parameter 2 has value of 6, indicates whether this is a system (0x0) or device (0x1) power IRP.	An error occurred during the handling of the power I/O request packet (IRP).
0x2	Reserved	Reserved	Reserved	An internal failure has occurred while attempting to process a power event. For more information, see Debugging bug check 0xA0 when parameter 1 equals 0x2.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE
0x3	The expected checksum	The actual checksum	The line number of the failure	The checksum for a hibernation context page does not match its expected checksum.
0x4	The expected checksum	The actual checksum	The line number of the failure	The checksum for a page about to be written to the hibernation file does not match its expected checksum.
0x5	Reserved	Reserved	Reserved	An unknown shutdown code has been sent to the system shutdown handler.
0x7	Reserved	Reserved	Reserved	An unhandled exception has occurred. For more information, see Debugging bug check 0xA0 when parameter 1 equals 0x7.
0x8	This parameter is always set to 0x100.	The device object	POWER_CHANNEL_SUMMARY	A fatal error occurred while processing a system power event.
0x9	Status code	Mirroring phase	Reserved	A fatal error occured while preparing the hibernate file.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE
0xA	**0:** A bug check was requested immediately upon resuming. **1:** A bug check was requested during resume after all non-pageable devices had been powered on. **2:** A bug check was requested during resume after all devices had been powered on.	Reserved	Reserved	A bug check was requested when waking for debugging purposes.
0xB	Size of the hibernation file.	Hibernation progress before running out of space **0:** HIBERFILE_PROGRESS_FREE_MAP **1:** HIBERFILE_PROGRESS_RESUME_CONTEXT **2:** HIBERFILE_PROGRESS_PROCESSOR_STATEE **3:** HIBERFILE_PROGRESS_MEMORY_RANGES **4:** HIBERFILE_PROGRESS_TABLE_PAGES **5:** HIBERFILE_PROGRESS_MEMORY_IMAGE	Size of the remaining memory ranges.	The hibernation file is too small.
0xC	Status code	Dump stack context	Reserved	The dump stack failed to initialize.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE
0x101	Reserved	Exception pointer.	Reserved	An unhandled exception occured while processing a system power event. For more information, see Debugging bug check 0xA0 when parameter 1 equals 0x101.
0x102	Reserved	DUMP_INITIALIZATION_CONTEXT	POP_HIBER_CONTEXT	The hibernation working buffer size is not page aligned.
0x103	Reserved	POP_HIBER_CONTEXT	Reserved	All working pages have failed to be accounted for during the hibernation process.
0x104	Reserved	POP_HIBER_CONTEXT	Reserved	An attempt was made to map internal hibernation memory while the internal memory structures were locked.
0x105	Reserved	POP_HIBER_CONTEXT	Reserved	An attempt was made to map internal hibernation memory with an unsupported memory type flag.
0x106	Reserved	The memory descriptor list (MDL)	Reserved	A memory descriptor list was created during the hibernation process which describes memory that is not paged-aligned.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE
0x107	Reserved	POP_HIBER_CONTEXT	PO_MEMORY_RANGE_ARRAY	A data mismatch has occurred in the internal hibernation data structures.
0x108	Reserved	POP_HIBER_CONTEXT	Reserved	The disk subsystem failed to properly write part of the hibernation file.
0x109	Reserved	Expected checksum	Actual checksum	The checksum for the processor state data does not match its expected checksum.
0x10A	Reserved	POP_HIBER_CONTEXT	NTSTATUS	The disk subsystem failed to properly read or write part of the hibernation file.
0x10B	Reserved	Current hibernation progress	Reserved	An attempt was made to mark pages for the boot phase of hibernation at the wrong time using the PoSetHiberRange API.
0x10C	Reserved	Flags provided to the API	Length to mark	The PoSetHiberRange API was called with invalid parameters.
0x200	Reserved	DEVICE_OBJECT	DEVICE_OBJECT_POWER_EXTENSION	An unknown device type is being checked for an idle state.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE
0x300	Reserved	DEVICE_OBJECT	IRP	An unknown status was returned from a battery power IRP.
0x301	Reserved	DEVICE_OBJECT	IRP	The battery has entered an unknown state.
0x400	Reserved	IO_STACK_LOCATION	DEVICE_OBJECT	A device has overrun its maximum number of reference counts.
0x401	Reserved	Pending IRP list	DEVICE_OBJECT	Too many inrush power IRPs have been queued.
0x402	Reserved	Pending IRP list	DEVICE_OBJECT	Too many inrush power IRPs have been queued.
0x403	Reserved	Pending IRP list	DEVICE_OBJECT	Too many inrush power IRPs have been queued.
0x404	Reserved	IO_STACK_LOCATION	DEVICE_OBJECT	A power IRP has been sent to a passive-level device object.
0x500	Reserved	IRP	DEVICE_OBJECT	An unknown status was returned from a thermal power IRP.
0x600	DEVICE_OBJECT PDO	Reserved	Reserved	A driver has attempted a duplicate registration with the Power Runtime Framework.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE
0x601	POP_FX_DEVICE device	PEP_DEVICE_REGISTER PEP	Reserved	No Power Engine Plugins accepted device registration.
0x602	DEVICE_NODE device node	Sleep count	Reserved	Device node sleep count does not match its activation count.
0x603	POP_FX_PLUGIN	Work request type	Reserved	A Power Engine Plugin made an invalid work request.
0x605	Notification ID	POP_FX_PLUGIN	Reserved	A Power Engine Plugin failed to accept mandatory device power management notification.
0x606	POP_FX_COMPONENT	POP_FX_COMPONENT_FLAGS	New condition for the component	A Power Engine Plugin attempted to transition a critical system resource component to an Active (or Idle) condition when the resource was already Active (or Idle).
0x607	POP_FX_DEVICE	NTSTATUS	Reserved	The acquisition of a runtime power management framework device-removal lock failed when it was required to succeed.
0x608	POP_FX_COMPONENT	POP_FX_COMPONENT_FLAGS	Reserved	A driver has attempted to transition a component to idle without a preceding active request.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE
0x609	POP_FX_PLUGIN	POP_FX_DEVICE	Duplicate Request Type **0:** DevicePowerRequired **1:** DevicePowerNotRequired	A Power Engine Plugin has requested either device power required or device power not required without an intervening request of the opposite type.
0x610	POP_FX_PLUGIN	POP_FX_DEVICE	Reserved	A Power Engine Plugin has requested device power not required while a previous device power required request is outstanding.
0x611	POP_FX_PLUGIN	POP_FX_DEVICE	Invalid component index	A Power Engine Plugin has requested an operation on an invalid component.
0x612	POP_FX_PLUGIN PowerEnginePlugin	Reserved	Reserved	A Power Engine Plugin has requested additional work to be done in the context of a device notification where no buffer was supplied by PO for the request.
0x613	POP_FX_DEVICE	Component index	Operation **0:** Complete device power not required **1:** Report device powered on **2:** Complete idle condition	A driver has attempted to complete a request when no such outstanding request is pending.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE
0x614	POP_FX_DEVICE	Component index	Illegal parameter **0:** PO_FX_FLAG_BLOCKING used at IRQL >= DISPATCH_LEVEL **1:** PO_FX_FLAG_BLOCKING and PO_FX_FLAG_ASYNC_ONLY both specified **2:** Invalid component index	A driver has requested an active/idle transition on a component with an illegal parameter.
0x615	POP_FX_PLUGIN	POP_FX_COMPONENT	Illegal Action **0:** Component not in idle state 0 **1:**Component is already active **2:** No outstanding activation request **3:** Outstanding idle state transition	A Power Engine Plugin has illegally indicated the completion of a component activation.
0x616	POP_FX_PLUGIN	POP_FX_COMPONENT	Illegal Action **0:** Invalid idle state **1:** Component is already in the requested state **2:** Requested a non-zero idle state without passing through idle state 0	A Power Engine Plugin has illegally requested a component idle state transition.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE
0x666	PPOP_PEP_ACTIVITY	New activity type **0:** DevicePowerOn **1:** ComponentIdleStateChange **2:** ComponentActivating **3:** ComponentActive **4:** DevicePowerOff **5:** DeviceSuspend	Conflicting activity type **0:** DevicePowerOn **1:** ComponentIdleStateChange **2:** ComponentActivating **3:** ComponentActive **4:** DevicePowerOff **5:** DeviceSuspend	The default Power Engine Plugin has attempted to trigger a new activity that conflicts with another activity.
0x667	POP_PEP_ACTIVITY	Activity type **0:** DevicePowerOn **1:** ComponentIdleStateChange **2:** ComponentActivating **3:** ComponentActive **4:** DevicePowerOff **5:** DeviceSuspend	POP_PEP_ACTIVITY_STATUS	Default Power Engine Plugin has attempted to complete an activity that is not running.
0x700	PEPHANDLE	PEP_PPM_IDLE_SELECT	Reserved	A Power Engine Plugin has specified invalid processor idle dependencies.
0x701	The index of the selected idle state of the hung processor	The PRCB address of the hung processor	The index of the hung processor	A processor was not able to complete an idle transition within the allocated interval. This indicates the specified processor is hung.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE
0x702	The index of the selected idle state of the processor	The idle synchronization state of the processor	The PRCB address of the hung processor	A processor woke up from a non-interruptible state without the OS initiating an explicit wake through the PEP (using the necessary PPM idle synchronization).

Resolution

General Notes

In the preceding table, several of the parameters are pointers to structures. For example, if Parameter 2 is listed as DEVICE_OBJECT, then Parameter 2 is a pointer to a DEVICE_OBJECT structure. Some of the structures are defined in wdm.h, which is included in the Windows Driver Kit. For example, the following structures are defined in wdm.h.

- EXCEPTION_POINTERS
- DEVICE_OBJECT
- IO_STACK_LOCATION
- PEP_DEVICE_REGISTER

Some of the structures that appear in the preceding table are not defined in any public header file. You can see the definitions of those structures by using the **dt** debugger command. The following example shows how to use the **dt** command to see the **DEVICE_OBJECT_POWER_EXTENSION** structure.

```
3: kd> dt nt!DEVICE_OBJECT_POWER_EXTENSION
   +0x000 IdleCount        : Uint4B
   +0x004 BusyCount        : Uint4B
   +0x008 BusyReference    : Uint4B
   +0x00c TotalBusyCount   : Uint4B
   +0x010 ConservationIdleTime : Uint4B
   +0x014 PerformanceIdleTime : Uint4B
   +0x018 DeviceObject     : Ptr64 _DEVICE_OBJECT
   +0x020 IdleList         : _LIST_ENTRY
   +0x030 IdleType         : _POP_DEVICE_IDLE_TYPE
   +0x034 IdleState        : _DEVICE_POWER_STATE
   +0x038 CurrentState     : _DEVICE_POWER_STATE
   +0x040 Volume           : _LIST_ENTRY
   +0x050 Specific         : <unnamed-tag>
```

The following procedures will help you debug certain instances of this bug check.

Debugging bug check 0xA0 when Parameter 1 equals 0x2

1. Examine the stack. Look for the **ntoskrnl!PopExceptionFilter** function. This function contains the following code as its first argument.

   ```
   (error_code << 16) | _LINE_
   ```

   If the caller is **PopExceptionFilter**, the first argument to this function is of type PEXCEPTION_POINTERS.

Note the value of this argument.

2. Use the **dt (Display Type)** command and specify the value that you found in the previous step as *argument*.

```
dt nt!_EXCEPTION_POINTERS argument
```

This command displays the structure. Note the address of the context record.

3. Use the **.cxr (Display Context Record)** command and specify the context record that you found in the previous step as *record*.

```
.cxr record
```

This command sets the register context to the proper value.

4. Use a variety of commands to analyze the source of the error. Start with **kb (Display Stack Backtrace)** .

Debugging bug check 0xA0 when Parameter 1 equals 0x7

1. Examine the stack. Look for the **ntoskrnl!PopExceptionFilter** function. The first argument to this function is of type PEXCEPTION_POINTERS. Note the value of this argument.

2. Use the **dt (Display Type)** command and specify the value that you found in the previous step as *argument*.

```
dt nt!_EXCEPTION_POINTERS argument
```

This command displays the structure. Note the address of the context record.

3. Use the **.cxr (Display Context Record)** command and specify the context record that you found in the previous step as *record*.

```
.cxr record
```

This command sets the register context to the proper value.

4. Use a variety of commands to analyze the source of the error. Start with **kb (Display Stack Backtrace)** .

Debugging bug check 0xA0 when Parameter 1 equals 0x101

1. Use the **dt (Display Type)** command and specify the value of Parameter 3 as *argument*.

```
dt nt!_EXCEPTION_POINTERS argument
```

This command displays the structure. Note the address of the context record.

2. Use the **.cxr (Display Context Record)** command and specify the context record that you found the previous step as *record*.

```
.cxr record
```

This command sets the register context to the proper value.

3. Use a variety of commands to analyze the source of the error. Start with **kb (Display Stack Backtrace)** .

Bug Check 0xA1: PCI_BUS_DRIVER_INTERNAL

7/2/2019 • 2 minutes to read • Edit Online

The PCI_BUS_DRIVER_INTERNAL bug check has a value of 0x000000A1. This bug check indicates that the PCI Bus driver detected inconsistency problems in its internal structures and could not continue.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

PCI_BUS_DRIVER_INTERNAL Parameters

None

Bug Check 0xA2: MEMORY_IMAGE_CORRUPT

7/2/2019 • 2 minutes to read • Edit Online

The MEMORY_IMAGE_CORRUPT bug check has a value of 0x000000A2. This bug check indicates that corruption has been detected in the image of an executable file in memory.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

MEMORY_IMAGE_CORRUPT Parameters

Parameter 1 indicates the type of violation. The meaning of the other parameters depends on the value of Parameter 1.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE
0x02	**If Parameter 3 is zero:** The page number in the table page that failed **If Parameter 3 is nonzero:** The page number with the failing page run index	Zero, or the index that failed to match the run	0	A table page check failure occurred.
0x03	The starting physical page number of the range	The length (in pages) of the range	The page number of the table page that contains this run	The checksum for the range of memory listed is incorrect.

Cause

A cyclic redundancy check (CRC) check on the memory range has failed.

On a system wake operation, various regions of memory might be checked to guard against memory failures.

Bug Check 0xA3: ACPI_DRIVER_INTERNAL

7/2/2019 • 2 minutes to read • Edit Online

The ACPI_DRIVER_INTERNAL bug check has a value of 0x000000A3. This bug check indicates that the ACPI driver detected an internal inconsistency.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

ACPI_DRIVER_INTERNAL Parameters

PARAMETER	DESCRIPTION
1	Reserved
2	Reserved
3	Reserved
4	Reserved

Cause

An inconsistency in the ACPI driver is so severe that continuing to run would cause serious problems.

One possible source of this problem is a BIOS error.

Bug Check 0xA4: CNSS_FILE_SYSTEM_FILTER

7/2/2019 • 2 minutes to read • Edit Online

The CNSS_FILE_SYSTEM_FILTER bug check has a value of 0x000000A4. This bug check indicates that a problem occurred in the CNSS file system filter.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

CNSS_FILE_SYSTEM_FILTER Parameters

PARAMETER	DESCRIPTION
1	Specifies source file and line number information. The high 16 bits (the first four hexadecimal digits after the "0x") identify the source file by its identifier number. The low 16 bits identify the source line in the file where the bug check occurred.
2	Reserved
3	Reserved
4	Reserved

Cause

The CNSS_FILE_SYSTEM_FILTER bug check might occur because nonpaged pool memory is full. If the nonpaged pool memory is completely full, this error can stop the system. However, during the indexing process, if the amount of available nonpaged pool memory is very low, another kernel-mode driver that requires nonpaged pool memory can also trigger this error.

Resolution

To resolve a nonpaged pool memory depletion problem: Add new physical memory to the computer. This memory sincrease the quantity of nonpaged pool memory available to the kernel.

Bug Check 0xA5: ACPI_BIOS_ERROR

9/16/2019 • 10 minutes to read • Edit Online

The ACPI_BIOS_ERROR bug check has a value of 0x000000A5. This bug check indicates that the Advanced Configuration and Power Interface (ACPI) BIOS of the computer is not fully compliant with the ACPI specification.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

ACPI_BIOS_ERROR Parameters

Parameter 1 indicates the kind of the incompatibility. The meaning of the other parameters depends on the value of Parameter 1.

If the BIOS incompatibility is related to Plug and Play (PnP) or power management, the following parameters are used.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE
0x01	ACPI's **deviceExtension**	ACPI's **ResourceList**	**0:** No resource list is found **1:** No IRQ resource is found in list	ACPI cannot find the System Control Interrupt (SCI) vector in the resources that are handed to it when ACPI is started.
0x02				(See the table later on this page)
0x03	The ACPI object that was being run	The return value from the interpreter	The name of the control method (in ULONG format)	ACPI tried to run a control method while creating device extensions to represent the ACPI namespace, but this control method failed.
0x04	The ACPI extension that _PRW belongs to	A pointer to the method	The **DataType** returned (see Amli.h)	ACPI evaluated a _PRW and expected to find an integer as a package element.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE
0x05	The ACPI extension that _PRW belongs to	Aointer to the _PRW	The number of elements in the _PRW	ACPI evaluated a _PRW, and the package that came back failed to contain at least two elements. The ACPI specification requires that two elements always be present in a _PRW.
0x06	The ACPI extension that _PRx belongs to	A pointer to the _PRx	A pointer to the name of the object to look for	ACPI tried to find a named object, but it could not find the object.
0x07	The ACPI extension that the method belongs to	A pointer to the method	The **DataType** returned (see Amli.h)	ACPI evaluated a method and expected to receive a buffer in return. However, the method returned some other data type.
0x08	The ACPI extension that the method belongs to	A pointer to the method	The **DataType** returned (see Amli.h)	ACPI evaluated a method and expected to receive an integer in return. However, the method returned some other data type.
0x09	The ACPI extension that the method belongs to	A pointer to the method	The **DataType** returned (see Amli.h)	ACPI evaluated a method and expected to receive a package in return. However, the method returned some other data type.
0x0A	The ACPI extension that the method belongs to	A pointer to the method	The **DataType** returned (see Amli.h)	ACPI evaluated a method and expected to receive a string in return. However, the method returned some other data type.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE
0x0B	The ACPI extension that _EJD belongs to	The status that the interpreter returns	The name of the object that ACPI is trying to find	ACPI cannot find the object that an _EJD string references.
0x0C	The ACPI extension that ACPI found a dock device for	A pointer to the _EJD method	**0:** BIOS does not claim system is dockage **1:** Duplicate device extensions for dock device	ACPI provides faulty or insufficient information for dock support.
0x0D	The ACPI extension that ACPI needs the object for	The (ULONG) name of the method that ACPI looked for	**0:** Base case **1:** Conflict	ACPI could not find a required method or object in the namespace This bug check code is used if there is no _HID or _ADR present.
0x0E	The NS **PowerResource** that ACPI needs the object for	The (ULONG) name of the method that ACPI looked for	0: Base case	ACPI could not find a required method or object in the namespace for a power resource (or entity other than a "device"). This bug check code is used if there is no _ON, _OFF, or _STA present for a power resource.
0x0F	The current buffer that ACPI was parsing	The buffer's tag	The specified length of the buffer	ACPI could not parse the resource descriptor.
0x10				(See the table later on this page)
0x11				(See the table later on this page)

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE
0x14	The current buffer that ACPI was parsing	The buffer's tag	A pointer to a variable that contains the ULONGLONG length of the buffer	ACPI could not parse the resource descriptor. The length exceeds MAXULONG.
0x15	The ACPI Machine Language (AML) context	**1:** Failed to load table **2:** The Parameter Path String Object was not found **3:** Failed to insert Parameter Data into the ParameterPath String Object **4:** Out of system memory	The NT status code	ACPI had a fatal error when attempting to load a table.
0x16	A pointer to the parent NSOBJ	A pointer to the illegal child ACPI namespace object	Reserved	ACPI had a fatal error when processing an xSDT. An object was declared as a child of a parent that cannot have children.

If an interrupt routing failure or incompatibility has occurred, the following parameters are used.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE
0x2001	**InterruptModel** (integer)	The return value from the interpreter	A pointer to the PIC control method	ACPI tried to evaluate the PIC control method but failed.
0x10001	A pointer to the device object	A pointer to the parent of the device object	A pointer to the _PRT object (See the following Comments section)	ACPI tried to do interrupt routing, but failed.
0x10002	A pointer to the device object	A pointer to the string name that ACPI was looking for but could not find	A pointer to the _PRT object (See the following Comments section)	ACPI could not find the link node referenced in a _PRT.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE
0x10003	A pointer to the device object	The device ID or function number. This DWORD is encoded as follows: bits 5:0 are the PCI device number, and bits 8:6 are the PCI function number	A pointer to the _PRT object (See the following Comments section)	ACPI could not find a mapping in the _PRT package for a device.
0x10005	A pointer to the _PRT object (See the following Comments section)	A pointer to the current _PRT element. (This pointer is an index into the _PRT.)	The device ID or function number. This DWORD is encoded as follows: bits 15:0 are the PCI function number, and bits 31:16 are the PCI device number	ACPI found an entry in the _PRT that the function ID is not all F's for. (The generic format for a _PRT entry is that the device number is specified, but the function number is not.)
0x10006	A pointer to the link node. (This device is missing the _DIS method.)	0	0	ACPI found a link node, but it cannot disable the node. (Link nodes must be disabled to allow for reprogramming.)
0x10007	The vector that could not be found	0	0	The _PRT contained a reference to a vector that is not described in the I/O APIC entry's MAPIC table.
0x10008	The invalid interrupt level.	0	0	The ACPI SCI interrupt level is invalid.
0x10009	0	0	0	The Fixed ACPI Description Table (FADT) could not be located.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE
0x1000A	0	0	0	The Root System Description Pointer (RSDP) or Extended System Description Table (XSDT) could not be located
0x1000B	The ACPI table signature	A pointer to the ACPI table	0	The length of the ACPI table is not consistent with the table revision.
0x1000C	Revision ID	Function Index	0	_DSM method for interrupts returned malformed data.
0x1000D	The ACPI Extension for the device	Value 0 : _PRW specified with no wake-capable interrupts and at least one GPIO interrupt Value 1 : Since there are wake-capable interrupts, _PRW should specify a GpeInfo value of 0xffffffff	0	A device used both GPE and GPIO interrupts, which is not supported.
0x1000E	The Status returned by the validation function.	Pointer to the ACPI namespace path UNICODE_STRING.	Pointer to the resource list compared against the SDEV.	A secure device's SDEV resources do not match its corresponding _CRS or _PRS entry.

If a miscellaneous failure or incompatibility has occurred, the following parameters are used.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE
0x20000	The I/O port in the Fixed Table	0	0	The PM_TMR_BLK entry in the Fixed ACPI Description Table does not point to a working ACPI timer block.

This table describes memory usage issues where the following parameters are used.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE
0x1000	The high portion of the physical address of the memory region.	The low portion of the physical address of the memory region.	The length of memory being mapped.	ACPI had a fatal error when processing a memory operation region. The memory operation region tried to map memory that has been allocated for OS usage.

If Parameter 1 equals **0x02**, the ACPI BIOS could not process the resource list for the PCI root buses. In this case, Parameter 3 specifies the exact problem, and the remaining parameters have the following definitions.

PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE
The ACPI extension for the PCI bus	0x0	A pointer to the QUERY_RESOURCES IRP	ACPI cannot convert the BIOS' resource list into the proper format. This probably represents an error in the BIOS' list encoding procedure.
The ACPI extension for the PCI bus	0x1	A pointer to the QUERY_RESOURCE_REQUIREMENTS IRP	ACPI cannot convert the BIOS' resource list into the proper format. This probably represents an error in the BIOS' list encoding procedure.
The ACPI extension for the PCI bus	0x2	0	ACPI found an empty resource list.
The ACPI extension for the PCI bus	0x3	A pointer to the PNP CRS descriptor	ACPI could not find the current bus number in the CRS.
The ACPI extension for the PCI bus	A pointer to the resource list for PCI	A pointer to the E820 memory table	The list of resources that PCI claims to decode overlaps with the list of memory regions that the E820 BIOS interface reports. (This kind of conflict is never permitted.)

If Parameter 1 equals **0x10**, the ACPI BIOS could not determine the system-to-device-state mapping correctly. In this situation, Parameter 3 specifies the exact problem, and the remaining parameters have the following definitions.

PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE
The ACPI extension whose mapping is needed	0x0	The DEVICE_POWER_STATE (this is "x+1")	_PRx was mapped back to a non-supported S-state.
The ACPI extension whose mapping is needed	0x1	The SYSTEM_POWER_STATE that cannot be mapped	ACPI cannot find a D-state to associate with the S-state.
The ACPI extension whose mapping is needed	0x2	The SYSTEM_POWER_STATE that cannot be mapped	The device claims to be able to wake the system when the system is in this S-state, but the system does not actually support this S-state.

If Parameter 1 equals **0x11**, the system could not enter ACPI mode. In this situation, Parameter 2 specifies the exact problem, and the remaining parameters have the following definitions.

PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE
0x0	0	0	The system could not initialize the AML interpreter.
0x1	0	0	The system could not find RSDT.
0x2	0	0	The system could not allocate critical driver structures.
0x3	0	0	The system could not load RSDT.
0x4	0	0	The system could not load DDBs.
0x5	0	0	The system cannot connect the Interrupt vector.
0x6	0	0	SCI_EN never becomes set in PM1 Control Register.

PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE
0x7	A pointer to the table that had a bad checksum	Creator revision	The table checksum is incorrect.
0x8	A pointer to the table that ACPI failed to load	Creator revision	ACPI failed to load DDB.
0x9	FADT version	0	Unsupported firmware version.
0xA	0	0	The system could not find MADT.
0xB	0	0	The system could not find any valid Local SAPIC structures in the MADT.

Cause

The value of Parameter 1 indicates the error.

Resolution

If you are debugging this error, use the **!analyze -v** extension. This extension displays all the relevant data (device extensions, nsobjects, or whatever is appropriate to the specific error).

If you are not performing debugging, this error indicates that you have to obtain a new BIOS. Contact your vendor or visit the internet to get a new BIOS.

If you cannot obtain an updated BIOS, or the latest BIOS is still not ACPI compliant, you can turn off ACPI mode during text-mode setup. To turn off ACPI mode, press the F7 key when you are prompted to install storage drivers. The system does not notify you that the F7 key was pressed, but it silently disables ACPI and enables you to continue your installation.

Remarks

A PCI routing table (_PRT) is the ACPI BIOS object that specifies how all the PCI devices are connected to the interrupt controllers. A computer with multiple PCI buses might have multiple _PRTs.

You can display a _PRT in the debugger by using the **!acpikd.nsobj** extension together with the address of the _PRT object as its argument.

Bug Check 0xA7: BAD_EXHANDLE

7/2/2019 • 2 minutes to read • Edit Online

The BAD_EXHANDLE bug check has a value of 0x000000A7. This bug check indicates that the kernel-mode handle table detected an inconsistent handle table entry state.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

BAD_EXHANDLE Parameters

None

Bug Check 0xAB: SESSION_HAS_VALID_POOL_ON_EXIT

7/2/2019 • 2 minutes to read • Edit Online

The SESSION_HAS_VALID_POOL_ON_EXIT bug check has a value of 0x000000AB. This bug check indicates that a session unload occurred while a session driver still held memory.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

SESSION_HAS_VALID_POOL_ON_EXIT Parameters

PARAMETER	DESCRIPTION
1	The session ID.
2	The number of paged pool bytes that are leaking.
3	The number of nonpaged pool bytes that are leaking.
4	The total number of paged and nonpaged allocations that are leaking. (The number of nonpaged allocations are in the upper half of this word, and paged allocations are in the lower half of this word.)

Cause

The SESSION_HAS_VALID_POOL_ON_EXIT bug check occurs because a session driver does not free its pool allocations before a session unload. This bug check can indicate a bug in Win32k.sys, Atmfd.dll, Rdpdd.dll, or a video or other driver.

Bug Check 0xAC: HAL_MEMORY_ALLOCATION

7/2/2019 • 2 minutes to read • Edit Online

The HAL_MEMORY_ALLOCATION bug check has a value of 0x000000AC. This bug check indicates that the hardware abstraction layer (HAL) could not obtain sufficient memory.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

HAL_MEMORY_ALLOCATION Parameters

PARAMETER	DESCRIPTION
1	The allocation size
2	0
3	A pointer to a string that contains the file name
4	Reserved

Cause

The HAL could not obtain non-paged memory pool for a system critical requirement.

These critical memory allocations are made early in system initialization, and the HAL_MEMORY_ALLOCATION bug check is not expected. This bug check probably indicates some other critical error such as pool corruption or massive consumption.

Bug Check 0xAD: VIDEO_DRIVER_DEBUG_REPORT_REQUEST

7/2/2019 • 2 minutes to read • Edit Online

The VIDEO_DRIVER_DEBUG_REPORT_REQUEST bug check has a value of 0x000000AD. This bug check indicates that the video port created a non-fatal minidump on behalf of the video driver during run time.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

VIDEO_DRIVER_DEBUG_REPORT_REQUEST Parameters

PARAMETER	DESCRIPTION
1	Driver-specific
2	Driver-specific
3	Driver-specific
4	The number of all reports that have been requested since boot time

Remarks

The video port created a non-fatal minidump on behalf of the video driver during run time because the video driver requested a debug report.

The VIDEO_DRIVER_DEBUG_REPORT_REQUEST bug check can be caused only by minidump creation, not by the creation of a full dump or kernel dump.

Bug Check 0xB1: BGI_DETECTED_VIOLATION

7/2/2019 • 2 minutes to read • Edit Online

The BGI_DETECTED_VIOLATION bug check has a value of 0x000000B1.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

BGI_DETECTED_VIOLATION Parameters

None

Bug Check 0xB4: VIDEO_DRIVER_INIT_FAILURE

7/2/2019 • 2 minutes to read • Edit Online

The VIDEO_DRIVER_INIT_FAILURE bug check has a value of 0x000000B4. This indicates that Windows was unable to enter graphics mode.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

VIDEO_DRIVER_INIT_FAILURE Parameters

None

Cause

The system was not able to go into graphics mode because no display drivers were able to start.

This usually occurs when no video miniport drivers are able to load successfully.

Bug Check 0xB8: ATTEMPTED_SWITCH_FROM_DPC

7/2/2019 • 2 minutes to read • Edit Online

The ATTEMPTED_SWITCH_FROM_DPC bug check has a value of 0x000000B8. This indicates that an illegal operation was attempted by a delayed procedure call (DPC) routine.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

ATTEMPTED_SWITCH_FROM_DPC Parameters

PARAMETER	DESCRIPTION
1	The original thread causing the failure
2	The new thread
3	The stack address of the original thread
4	Reserved

Cause

A wait operation, attach process, or yield was attempted from a DPC routine. This is an illegal operation.

Resolution

The stack trace will lead to the code in the original DPC routine that caused the error.

Bug Check 0xB9: CHIPSET_DETECTED_ERROR

7/13/2019 • 2 minutes to read • Edit Online

The CHIPSET_DETECTED_ERROR bug check has a value of 0x000000B9.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Resolution

The **!analyze** debug extension displays information about the bug check and can be very helpful in determining the root cause.

Bug Check 0xBA: SESSION_HAS_VALID_VIEWS_ON_EXIT

7/2/2019 • 2 minutes to read • Edit Online

The SESSION_HAS_VALID_VIEWS_ON_EXIT bug check has a value of 0x000000BA. This indicates that a session driver still had mapped views when the session unloaded.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

SESSION_HAS_VALID_VIEWS_ON_EXIT Parameters

PARAMETER	DESCRIPTION
1	The session ID
2	The number of mapped views that are leaking
3	The address of this session's mapped views table
4	The size of this session's mapped views table

Cause

This error is caused by a session driver not unmapping its mapped views prior to a session unload. This indicates a bug in win32k.sys, atmfd.dll, rdpdd.dll, or a video driver.

Bug Check 0xBB: NETWORK_BOOT_INITIALIZATION_FAILED

7/2/2019 • 2 minutes to read • Edit Online

The NETWORK_BOOT_INITIALIZATION_FAILED bug check has a value of 0x000000BB. This indicates that Windows failed to successfully boot off a network.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

NETWORK_BOOT_INITIALIZATION_FAILED Parameters

PARAMETER	DESCRIPTION
1	The part of network initialization that failed. Possible values are: **1:** Failure while updating the registry. **2:** Failure while starting the network stack. Windows sends IOCTLs to the redirector and datagram receiver, then waits for the redirector to be ready. If it is not ready within a certain period of time, this error is issued. **3:** Failure while sending the DHCP IOCTL to TCP. This is how Windows informs the transport of its IP address.
2	The failure status
3	Reserved
4	Reserved

Cause

This error is caused when Windows is booting off a network, and a critical function fails during I/O initialization.

Bug Check 0xBC: NETWORK_BOOT_DUPLICATE_ADDRESS

7/2/2019 • 2 minutes to read • Edit Online

The NETWORK_BOOT_DUPLICATE_ADDRESS bug check has a value of 0x000000BC. This indicates that a duplicate IP address was assigned to this machine while booting off a network.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

NETWORK_BOOT_DUPLICATE_ADDRESS Parameters

PARAMETER	DESCRIPTION
1	The IP address, shown as a DWORD. An address of the form *aa.bb.cc.dd* will appear as 0xDDCCBBAA.
2	The hardware address of the other machine. (For an Ethernet connection, see the following note.)
3	The hardware address of the other machine. (For an Ethernet connection, see the following note.)
4	The hardware address of the other machine. (For an Ethernet connection, this will be zero.)

Note When Parameter 4 equals zero, this indicates an Ethernet connection. In that case, the MAC address will be stored in Parameter 2 and Parameter 3. An Ethernet MAC address of the form *aa-bb-cc-dd-ee-ff* will cause Parameter 2 to equal 0xAABBCCDD, and Parameter 3 to equal 0xEEFF0000.

Cause

This error indicates that when TCP/IP sent out an ARP for its IP address, it got a response from another machine indicating a duplicate IP address.

When Windows is booting off a network, this is a fatal error.

Bug Check 0xBD: INVALID_HIBERNATED_STATE

7/2/2019 • 2 minutes to read • Edit Online

The INVALID_HIBERNATED_STATE bug check has a value of 0x000000BD. This indicates that the hibernated memory image does not match the current hardware configuration. This bugcheck occurs when a system resumes from hibernate and discovers that the hardware has been changed while the system was hibernated.

IMPORTANT

This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

INVALID_HIBERNATED_STATE Parameters

PARAMETER	DESCRIPTION
1	Hardware that was invalid. 1 : Number of installed processors is less than before the hibernation Value in Param 2: Number of processors before hibernation Value in Param 3: Number of processors after hibernation
2	Per Parameter 1
3	Per Parameter 1
4	Reserved

Bug Check 0xBE: ATTEMPTED_WRITE_TO_READONLY_MEMORY

7/2/2019 • 2 minutes to read • Edit Online

The ATTEMPTED_WRITE_TO_READONLY_MEMORY bug check has a value of 0x000000BE. This is issued if a driver attempts to write to a read-only memory segment.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

ATTEMPTED_WRITE_TO_READONLY_MEMORY Parameters

PARAMETER	DESCRIPTION
1	Virtual address of attempted write
2	PTE contents
3	Reserved
4	Reserved

If the driver responsible for the error can be identified, its name is printed on the blue screen and stored in memory at the location (PUNICODE_STRING) **KiBugCheckDriver**.

Remarks

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause.

Bug Check 0xBF: MUTEX_ALREADY_OWNED

7/2/2019 • 2 minutes to read • Edit Online

The MUTEX_ALREADY_OWNED bug check has a value of 0x000000BF. This indicates that a thread attempted to acquire ownership of a mutex it already owned.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

MUTEX_ALREADY_OWNED Parameters

PARAMETER	DESCRIPTION
1	The address of the mutex
2	The thread that caused the error
3	0
4	Reserved

Bug Check 0xC1: SPECIAL_POOL_DETECTED_MEMORY_CORRUPTION

7/2/2019 • 2 minutes to read • Edit Online

The SPECIAL_POOL_DETECTED_MEMORY_CORRUPTION bug check has a value of 0x000000C1. This indicates that the driver wrote to an invalid section of the special pool.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

SPECIAL_POOL_DETECTED_MEMORY_CORRUPTION Parameters

Parameter 4 indicates the type of violation.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
Address that the driver tried to free	Reserved	0	0x20	A driver attempted to free pool which was not allocated.
Address that the driver tried to free	Bytes requested	Bytes calculated (actually given to the caller)	0x21, 0x22	A driver attempted to free a bad address.
Address that the driver tried to free	Address where bits are corrupted	Reserved	0x23	A driver freed an address, but nearby bytes within the same page have been corrupted.
Address that the driver tried to free	Address where bits are corrupted	Reserved	0x24	A driver freed an address, but bytes occurring after the end of the allocation have been overwritten.
Current IRQL	Pool type	Number of bytes	0x30	A driver attempted to allocate pool at an incorrect IRQL.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
Current IRQL	Pool type	Address that the driver tried to free	0x31	A driver attempted to free pool at an incorrect IRQL.
Address that the driver tried to free	Address where one bit is corrupted	Reserved	0x32	A driver freed an address, but nearby bytes within the same page have a single bit error.

The _POOL_TYPE codes are enumerated in ntddk.h. In particular, zero indicates nonpaged pool and one indicates paged pool.

Cause

A driver has written to an invalid section of the special pool.

Resolution

Obtain a backtrace of the current thread. This backtrace will usually reveal the source of the error.

For information about the special pool, consult the Driver Verifier section of the Windows Driver Kit.

(Developer Content) Bug Check 0xC2: BAD_POOL_CALLER

7/2/2019 • 4 minutes to read • Edit Online

The BAD_POOL_CALLER bug check has a value of 0x000000C2. This indicates that the current thread is making a bad pool request.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

BAD_POOL_CALLER Parameters

Parameter 1 indicates the type of violation.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x00	0	Pool type	Pool tag	The current thread requested a zero-byte pool allocation.
0x01, 0x02, 0x04	Pointer to pool header	First part of pool header contents	0	The pool header has been corrupted.
0x06	Reserved	Pointer to pool header	Pool header contents	The current thread attempted to free the pool, which was already freed.
0x07	Reserved	Pool header contents	Address of the block of pool being freed	The current thread attempted to free the pool, which was already freed.
0x08	Current IRQL	Pool type	Size of allocation, in bytes	The current thread attempted to allocate the pool at an invalid IRQL.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x09	Current IRQL	Pool type	Address of pool	The current thread attempted to free the pool at an invalid IRQL.
0x0A	Address of pool	Allocator's tag	Tag being used in the attempted free	The current thread attempted to free pool memory by using the wrong tag. (The memory might belong to another component.)
0x0B, 0x0C, or 0x0D	Address of pool	Pool allocation's tag	Bad quota process pointer	The current thread attempted to release a quota on a corrupted pool allocation.
0x40	Starting address	Start of system address space	0	The current thread attempted to free the kernel pool at a user-mode address.
0x41	Starting address	Physical page frame	Highest physical page frame	The current thread attempted to free a non-allocated nonpaged pool address.
0x42 or 0x43	Address being freed	0	0	The current thread attempted to free a virtual address that was never in any pool.
0x44	Starting address	Reserved	0	The current thread attempted to free a non-allocated nonpaged pool address.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x46	Starting address	0	0	The current thread attempted to free an invalid pool address.
0x47	Starting address	Physical page frame	Highest physical page frame	The current thread attempted to free a non-allocated nonpaged pool address.
0x48	Starting address	Reserved	Reserved	The current thread attempted to free a non-allocated paged pool address.
0x50	Starting address	Start offset, in pages, from beginning of paged pool	Size of paged pool, in bytes	The current thread attempted to free a non-allocated paged pool address.
0x60	Starting address	0	0	The current thread attempted to free an invalid contiguous memory address. (The caller of **MmFreeContiguousMemory** is passing a bad pointer.)
0x99	Address that is being freed	0	0	The current thread attempted to free pool with an invalid address. (This code can also indicate corruption in the pool header.)

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x9A	Pool type	Number of bytes requested	Pool tag	The current thread marked an allocation request MUST_SUCCEED. (This pool type is no longer supported.)
0x9B	Pool type	Number of bytes requested	Caller's address	The current thread attempted to allocate a pool with a tag of 0 (This would be untrackable, and possibly corrupt the existing tag tables.)
0x9C	Pool type	Number of bytes requested	Caller's address	The current thread attempted to allocate a pool with a tag of "BIG". (This would be untrackable and could possibly corrupt the existing tag tables.)
0x9D	Incorrect pool tag used	Pool type	Caller's address	The current thread attempted to allocate a pool with a tag that does not contain any letters or digits. Using such tags makes tracking pool issues difficult.
0x41286	Reserved	Reserved	Start offset from the beginning of the paged pool, in pages	The current thread attempted to free a paged pool address in the middle of an allocation.

The _POOL_TYPE codes are enumerated in Ntddk.h. In particular, 0 indicates nonpaged pool and 1 indicates paged pool.

Cause

An invalid pool request has been made by the current thread. Typically this is at a bad IRQL level or double freeing the same memory allocation, etc.

Resolution

Activate Driver Verifier with memory pool options enabled, to obtain more information about these errors and to locate the faulting driver.

Driver Verifier

Driver Verifier is a tool that runs in real time to examine the behavior of drivers. If it see errors in the execution of driver code, it proactively creates an exception to allow that part of the driver code to be further scrutinized. The driver verifier manager is built into Windows and is available on all Windows PCs. To start the driver verifier manager, type *Verifer* at a command prompt. You can configure which drivers you would like to verify. The code that verifies drivers adds overhead as it runs, so try and verify the smallest number of drivers as possible. For more information, see Driver Verifier.

Windows Memory Diagnostics

In particular, for situations with memory pool corruption, run the Windows Memory Diagnostics tool, to try and isolate the physical memory as a cause. In the control panel search box, type Memory, and then click **Diagnose your computer's memory problems**. After the test is run, use Event viewer to view the results under the System log. Look for the *MemoryDiagnostics-Results* entry to view the results.

Bug Check 0xC4: DRIVER_VERIFIER_DETECTED_VIOLATION

10/23/2019 • 39 minutes to read • Edit Online

The DRIVER_VERIFIER_DETECTED_VIOLATION bug check has a value of 0x000000C4. This is the general bug check code for fatal errors found by Driver Verifier. For more information, see Handling a Bug Check When Driver Verifier is Enabled.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

DRIVER_VERIFIER_DETECTED_VIOLATION Parameters

Parameter 1 identifies the type of violation. The meaning of the remaining parameters varies with the value of Parameter 1. The parameter values are described in the following table.

Note If you have trouble viewing all 5 columns in this table, try the following:

- Expand your browser window to full size.
- Place the cursor in the table and use the arrow keys to scroll left and right.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x00	Current IRQL	Pool type	0	The driver requested a zero-byte pool allocation.
0x01	Current IRQL	Pool type	Size of allocation, in bytes	The driver attempted to allocate paged memory with IRQL > APC_LEVEL.
0x02	Current IRQL	Pool type	Size of allocation, in bytes	The driver attempted to allocate nonpaged memory with IRQL > DISPATCH_LEVEL.
0x10	Bad Address	0	0	The driver attempted to free an address that was not returned from an allocate call.
0x11	Current IRQL	Pool type	Address of pool	The driver attempted to free paged pool with IRQL > APC_LEVEL.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x12	Current IRQL	Pool type	Address of pool	The driver attempted to free nonpaged pool with IRQL > DISPATCH_LEVEL.
0x13 or 0x14	Reserved	Pointer to pool header	Pool header contents	The driver attempted to free memory pool which was already freed.
0x16	Reserved	Pool address	0	The driver attempted to free pool at a bad address, or the driver passed invalid parameters to a memory routine.
0x30	Current IRQL	Requested IRQL	0	The driver passed an invalid parameter to KeRaiseIrql. (The parameter was either a value lower than the current IRQL, or a value higher than HIGH_LEVEL. This may be the result of using an uninitialized parameter.)
0x31	Current IRQL	Requested IRQL	0: New IRQL is bad 1: New IRQL is invalid inside a DPC routine	The driver passed an invalid parameter to KeLowerIrql. (The parameter was either a value higher than the current IRQL, or a value higher than HIGH_LEVEL. This may be the result of using an uninitialized parameter.)
0x32	Current IRQL	Spin lock address	0	The driver called KeReleaseSpinLock at an IRQL other than DISPATCH_LEVEL. (This may be due to a double-release of a spin lock.)
0x33	Current IRQL	Fast mutex address	0	The driver attempted to acquire fast mutex with IRQL > APC_LEVEL.
0x34	Current IRQL	Fast mutex address	0	The driver attempted to release fast mutex at an IRQL other than APC_LEVEL.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x35	Current IRQL	Spin lock address	Old IRQL	The kernel released a spin lock with IRQL not equal to DISPATCH_LEVEL.
0x36	Current IRQL	Spin lock number	Old IRQL	The kernel released a queued spin lock with IRQL not equal to DISPATCH_LEVEL.
0x37	Current IRQL	Thread APC disable count	Resource	The driver tried to acquire a resource, but APCs are not disabled.
0x38	Current IRQL	Thread APC disable count	Resource	The driver tried to release a resource, but APCs are not disabled.
0x39	Current IRQL	Thread APC disable count	Mutex	The driver tried to acquire a mutex "unsafe" with IRQL not equal to APC_LEVEL on entry.
0x3A	Current IRQL	Thread APC disable count	Mutex	The driver tried to release a mutex "unsafe" with IRQL not equal to APC_LEVEL on entry.
0x3C	Handle passed to routine	Object type	0	The driver called ObReferenceObjectByHandle with a bad handle.
0x3D	0	0	Address of the bad resource	The driver passed a bad (unaligned) resource to ExAcquireResourceExclusive.
0x3E	0	0	0	The driver called KeLeaveCriticalRegion for a thread that is not currently in a critical region.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x3F	Object address	New object reference count. -1: dereference case 1: reference case	0	The driver applied ObReferenceObject to an object that has a reference count of zero, or the driver applied ObDereferenceObject to an object that has a reference count of zero.
0x40	Current IRQL	Spin lock address	0	The driver called KeAcquireSpinLockAtDpcLevel with IRQL < DISPATCH_LEVEL.
0x41	Current IRQL	Spin lock address	0	The driver called KeReleaseSpinLockFromDpcLevel with IRQL < DISPATCH_LEVEL.
0x42	Current IRQL	Spin lock address	0	The driver called KeAcquireSpinLock with IRQL > DISPATCH_LEVEL.
0x51	Base address of allocation	Address of the reference beyond the allocation	Number of charged bytes	The driver attempted to free memory after having written past the end of the allocation. A bug check with this parameter occurs only when the Pool Tracking option of Driver Verifier is active.
0x52	Base address of allocation	Reserved	Number of charged bytes	The driver attempted to free memory after having written past the end of the allocation. A bug check with this parameter occurs only when the Pool Tracking option of Driver Verifier is active.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x53, 0x54, or 0x59	Base address of allocation	Reserved	Reserved	The driver attempted to free memory after having written past the end of the allocation. A bug check with this parameter occurs only when the Pool Tracking option of Driver Verifier is active.
0x60	Bytes allocated from paged pool	Bytes allocated from nonpaged pool	Total number of allocations that were not freed	The driver is unloading without first freeing its pool allocations. A bug check with this parameter occurs only when the Pool Tracking option of Driver Verifier is active.
0x61	Bytes allocated from paged pool	Bytes allocated from nonpaged pool	Total number of allocations that were not freed	A driver thread is attempting to allocate pool memory while the driver is unloading. A bug check with this parameter occurs only when the Pool Tracking option of Driver Verifier is active.
0x62	Name of the driver	Reserved	Total number of allocations that were not freed, including both paged and nonpaged pool	The driver is unloading without first freeing its pool allocations. A bug check with this parameter occurs only when the Pool Tracking option of Driver Verifier is active.
0x70	Current IRQL	MDL address	Access mode	The driver called MmProbeAndLockPages with IRQL > DISPATCH_LEVEL.
0x71	Current IRQL	MDL address	Process address	The driver called MmProbeAndLockProcessPages with IRQL > DISPATCH_LEVEL.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x72	Current IRQL	MDL address	Process address	The driver called MmProbeAndLockSelectedPages with IRQL > DISPATCH_LEVEL.
0x73	Current IRQL	In 32-bit Windows: Low 32 bits of the physical address In 64-bit Windows: the 64-bit physical address	Number of bytes	The driver called MmMapIoSpace with IRQL > DISPATCH_LEVEL.
0x74	Current IRQL	MDL address	Access mode	The driver called MmMapLockedPages in kernel mode with IRQL > DISPATCH_LEVEL.
0x75	Current IRQL	MDL address	Access mode	The driver called MmMapLockedPages in user mode with IRQL > APC_LEVEL.
0x76	Current IRQL	MDL address	Access mode	The driver called MmMapLockedPagesSpecifyCache in kernel mode with IRQL > DISPATCH_LEVEL.
0x77	Current IRQL	MDL address	Access mode	The driver called MmMapLockedPagesSpecifyCache in user mode with IRQL > APC_LEVEL.
0x78	Current IRQL	MDL address	0	The driver called MmUnlockPages with IRQL > DISPATCH_LEVEL.
0x79	Current IRQL	Virtual address being unmapped	MDL address	The driver called MmUnmapLockedPages in kernel mode with IRQL > DISPATCH_LEVEL.
0x7A	Current IRQL	Virtual address being unmapped	MDL address	The driver called MmUnmapLockedPages in user mode with IRQL > APC_LEVEL.
0x7B	Current IRQL	Virtual address being unmapped	Number of bytes	The driver called MmUnmapIoSpace with IRQL > APC_LEVEL.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x7C	MDL address	MDL flags	0	The driver called MmUnlockPages, and passed an MDL whose pages were never successfully locked.
0x7D	MDL address	MDL flags	0	The driver called MmUnlockPages, and passed an MDL whose pages are from nonpaged pool. (These should never be unlocked.)
0x7E	Current IRQL	DISPATCH_LEVEL	0	The driver called MmAllocatePagesForMdl, MmAllocatePagesForMdlEx, or MmFreePagesFromMdl with IRQL > DISPATCH_LEVEL.
0x7F	Current IRQL	MDL address	MDL flags	The driver called BuildMdlForNonPagedPool and passed an MDL whose pages are from paged pool.
0x80	Current IRQL	Event address	0	The driver called KeSetEvent with IRQL > DISPATCH_LEVEL.
0x81	MDL address	MDL flags	0	The driver called MmMapLockedPages. (You should use MmMapLockedPagesSpecifyCache instead, with the BugCheckOnFailure parameter set to FALSE.)
0x82	MDL address	MDL flags	0	The driver called MmMapLockedPagesSpecifyCache with the BugCheckOnFailure parameter equal to TRUE. (This parameter should be set to FALSE.)

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x83	Start of physical address range to map	Number of bytes to map	First page frame number that isn't locked down	The driver called MmMapIoSpace without having locked down the MDL pages. The physical pages represented by the physical address range being mapped must have been locked down prior to making this call.
0x85	MDL address	Number of pages to map	First page frame number that isn't locked down	The driver called MmMapLockedPages without having locked down the MDL pages.
0x89	MDL address	Pointer to the non-memory page in the MDL	The non-memory page number in the MDL	An MDL is not marked as "I/O", but it contains non-memory page addresses.
0x91	Reserved	Reserved	Reserved	The driver switched stacks using a method that is not supported by the operating system. The only supported way to extend a kernel mode stack is by using KeExpandKernelStackAndCallout.
0xA0 (Windows Server 2003 and later operating systems only)	Pointer to the IRP making the read or write request	Device object of the lower device	Number of the sector in which the error was detected	A cyclic redundancy check (CRC) error was detected on a hard disk. A bug check with this parameter occurs only when the Disk Integrity Checking option of Driver Verifier is active.
0xA1 (Windows Server 2003 and later operating systems only)	Copy of the IRP making the read or write request. (The actual IRP has been completed.)	Device object of the lower device	Number of the sector in which the error was detected	A CRC error was detected on a sector (asynchronously). A bug check with this parameter occurs only when the Disk Integrity Checking option of Driver Verifier is active.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0xA2 (Windows Server 2003 and later operating systems only)	IRP making the read or write request, or a copy of this IRP	Device object of the lower device	Number of the sector in which the error was detected	The CRCDISK checksum copies don't match. This could be a paging error. A bug check with this parameter occurs only when the Disk Integrity Checking option of Driver Verifier is active.
0xB0 (Windows Vista and later operating systems only)	MDL address	MDL flags	Incorrect MDL flags	The driver called MmProbeAndLockPages for an MDL with incorrect flags. For example, the driver passed an MDL created by MmBuildMdlForNonPagedPool to MmProbeAndLockPages.
0xB1 (Windows Vista and later operating systems only)	MDL address	MDL flags	Incorrect MDL flags	The driver called MmProbeAndLockProcessPages for an MDL with incorrect flags. For example, the driver passed an MDL created by MmBuildMdlForNonPagedPool to MmProbeAndLockProcessPages.
0xB2 (Windows Vista and later operating systems only)	MDL address	MDL flags	Incorrect MDL flags	The driver called MmMapLockedPages for an MDL with incorrect flags. For example, the driver passed an MDL that is already mapped to a system address or that was not locked to MmMapLockedPages.
0xB3 (Windows Vista and later operating systems only)	MDL address	MDL flags	Missing MDL flags (at least one was expected)	The driver called MmMapLockedPages for an MDL with incorrect flags. For example, the driver passed an MDL that is not locked to MmMapLockedPages.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0xB4 (Windows Vista and later operating systems only)	MDL address	MDL flags	Unexpected partial MDL flag	The driver called MmUnlockPages for a partial MDL. A partial MDL is one that was created by IoBuildPartialMdl.
0xB8 (Windows Vista and later operating systems only)	MDL address	MDL flags	Reserved	The pages that are described by the MDL are still mapped. The driver must unmap the pages before calling IoFreeMdl.
0xC0 (Windows Vista and later operating systems only)	Address of the IRP	Reserved	Reserved	The driver called IoCallDriver with interrupts disabled.
0xC1 (Windows Vista and later operating systems only)	Address of the driver dispatch routine	Reserved	Reserved	A driver dispatch routine was returned with interrupts disabled.
0xC2 (Windows Vista and later operating systems only)	Reserved	Reserved	Reserved	The driver called a Fast I/O dispatch routine after interrupts were disabled.
0xC3 (Windows Vista and later operating systems only)	Address of the driver Fast I/O dispatch routine	Reserved	Reserved	A driver Fast I/O dispatch routine was returned with interrupts disabled.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0xC5 (Windows Vista and later operating systems only)	Address of the driver dispatch routine	The current thread's APC disable count	The thread's APC disable count prior to calling the driver dispatch routine	A driver dispatch routine has changed the thread's APC disable count. The APC disable count is decremented each time a driver calls KeEnterCriticalRegion, FsRtlEnterFileSystem, or acquires a mutex. The APC disable count is incremented each time a driver calls KeLeaveCriticalRegion, KeReleaseMutex, or FsRtlExitFileSystem. Because these calls should always be in pairs, the APC disable count should be zero whenever a thread is exited. A negative value indicates that a driver has disabled APC calls without re-enabling them. A positive value indicates that the reverse is true.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0xC6 (Windows Vista and later operating systems only)	Address of the driver Fast I/O dispatch routine	Current thread's APC disable count	The thread's APC disable count prior to calling the Fast I/O driver dispatch routine	A driver Fast I/O dispatch routine has changed the thread's APC disable count. The APC disable count is decremented each time a driver calls KeEnterCriticalRegion, FsRtlEnterFileSystem, or acquires a mutex. The APC disable count is incremented each time a driver calls KeLeaveCriticalRegion, KeReleaseMutex, or FsRtlExitFileSystem. Because these calls should always be in pairs, the APC disable count should be zero whenever a thread is exited. A negative value indicates that a driver has disabled APC calls without re-enabling them. A positive value indicates that the reverse is true.
0xCA (Windows Vista and later operating systems only)	Address of the lookaside list	Reserved	Reserved	The driver has attempted to re-initialize a lookaside list.
0xCB (Windows Vista and later operating systems only)	Address of the lookaside list	Reserved	Reserved	The driver has attempted to delete an uninitialized lookaside list.
0xCC (Windows Vista and later operating systems only)	Address of the lookaside list	Starting address of the pool allocation	Size of the pool allocation	The driver has attempted to free a pool allocation that contains an active lookaside list.
0xCD (Windows Vista and later operating systems only)	Address of the lookaside list	Block size specified by the caller	Minimum supported block size	The driver has attempted to create a lookaside list with an allocation block size that is too small.
0xD0 (Windows Vista and later operating systems only)	Address of the ERESOURCE structure	Reserved	Reserved	The driver has attempted to re-initialize an ERESOURCE structure.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0xD1 (Windows Vista and later operating systems only)	Address of the ERESOURCE structure	Reserved	Reserved	The driver has attempted to delete an uninitialized ERESOURCE structure.
0xD2 (Windows Vista and later operating systems only)	Address of the ERESOURCE structure	Starting address of the pool allocation	Size of the pool allocation	The driver has attempted to free a pool allocation that contains an active ERESOURCE structure.
0xD5 (Windows Vista and later operating systems only)	Address of the IO_REMOVE_LOCK structure created by the checked build version of the driver	Current IoReleaseRemoveLock tag	Reserved	The current IoReleaseRemoveLock tag does not match the previous IoAcquireRemoveLock tag. If the driver calling IoReleaseRemoveLock is not in a checked build, Parameter 2 is the address of the shadow IO_REMOVE_LOCK structure created by Driver Verifier on behalf of the driver. In this case, the address of the IO_REMOVE_LOCK structure used by the driver is not used at all, because Driver Verifier is replacing the lock address for all the remove lock APIs. A bug check with this parameter occurs only when the I/O Verification option of Driver Verifier is active.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0xD6 (Windows Vista and later operating systems only)	Address of the IO_REMOVE_LOCK structure created by the checked build version of the driver	Tag that does not match previous IoAcquireRemoveLock tag	Previous IoAcquireRemoveLock tag	The current IoReleaseRemoveLockAndWait tag does not match the previous IoAcquireRemoveLock tag. If the driver calling IoReleaseRemoveLock is not a checked build, Parameter 2 is the address of the shadow IO_REMOVE_LOCK structure created by Driver Verifier on behalf of the driver. In this case, the address of the IO_REMOVE_LOCK structure used by the driver is not used at all, because Driver Verifier is replacing the lock address for all the remove lock APIs. A bug check with this parameter occurs only when the I/O Verification option of Driver Verifier is active.
0xD7 (Windows 7 operating systems and later only)	Address of the checked build Remove Lock structure that is used internally by Driver Verifier	Address of the Remove Lock structure that is specified by the driver	Reserved	A Remove Lock cannot be re-initialized, even after it calls IoReleaseRemoveLockAndWait, because other threads might still be using that lock (by calling IoAcquireRemoveLock). The driver should allocate the Remove Lock inside its device extension, and initialize it a single time. The lock will be deleted together with the device extension.
0xDA (Windows Vista and later operating systems only)	Starting address of the driver	WMI callback address inside the driver	Reserved	An attempt was made to unload a driver that has not deregistered its WMI callback function.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0xDB (Windows Vista and later operating systems only)	Address of the device object	Reserved	Reserved	An attempt was made to delete a device object that was not deregistered from WMI.
0xDC (Windows Vista and later operating systems only)	Reserved	Reserved	Reserved	An invalid RegHandle value was specified as a parameter of the function EtwUnregister.
0xDD (Windows Vista and later operating systems only)	Address of the call to EtwRegister	Starting address of the unloading driver	For Windows 8 and later versions, this parameter is the ETW RegHandle value.	An attempt was made to unload a driver without calling EtwUnregister.
0xDF (Windows 7 operating systems and later only)	Synchronization object address	The synchronization object is in session address space. Synchronization objects are not allowed in session address space because they can be manipulated from another session or from system threads that have no session virtual address space.		
0xE0 (Windows Vista and later operating systems only)	User-mode address that is used as a parameter	Size ,in bytes, of the address range that is used as a parameter	Reserved	A call was made to an operating system kernel function that specified a user-mode address as a parameter.
0xE1 (Windows Vista and later operating systems only)	Address of the synchronization object	Reserved	Reserved	A synchronization object was found to have an address that was either invalid or pageable.
0xE2 (Windows Vista and later operating systems only)	Address of the IRP	User-mode address present in the IRP	Reserved	An IRP with Irp->RequestorMode set to KernelMode was found to have a user-mode address as one of its members.
0xE3 (Windows Vista and later operating systems only)	Address of the call to the API	User-mode address used as a parameter in the API	Reserved	A driver has made a call to a kernel-mode ZwXxx routine with a user-mode address as a parameter.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0xE4 (Windows Vista and later operating systems only)	Address of the call to the API	Address of the malformed UNICODE_STRING structure	Reserved	A driver has made a call to a kernel-mode ZwXxx routine with a malformed UNICODE_STRING structure as a parameter.
0xE5 (Windows Vista and later operating systems only)	Current IRQL	Reserved	Reserved	A call was made to a Kernel API at the incorrect IRQL.
0xE6 (Windows Vista and later operating systems only)	Address inside the driver making the Zw API call	Current IRQL	Special kernel APCs.	Kernel Zw API was not called at IRQL = PASSIVE_LEVEL and with special kernel APCs enabled.
0xEA (Windows Vista and later operating systems only)	Current IRQL	The thread's APC disable count	Address of the pushlock	A driver has attempted to acquire a pushlock while APCs are enabled.
0xEB (Windows Vista and later operating systems only)	Current IRQL	The thread's APC disable count	Address of the pushlock	A driver has attempted to release a pushlock while APCs are enabled.
0xF0 (Windows Vista and later operating systems only)	Address of the destination buffer	Address of the source buffer	Number of bytes to copy	A driver called the memcpy function with overlapping source and destination buffers.
0xF5 (Windows Vista and later operating systems only)	Address of the NULL handle	Object type	Reserved	A driver passed a NULL handle to ObReferenceObjectByHandle.
0xF6 (Windows 7 operating systems and later)	Handle value being referenced	Address of the current process	Address inside the driver that performs the incorrect reference	A driver references a user-mode handle as kernel mode.
0xF7 (Windows 7 operating systems and later)	Handle value specified by the caller	Object type specified by the caller	AccessMode specified by the caller	A driver is attempting a user-mode reference for a kernel handle in the context of the system process.
0xFA (Windows 7 operating systems and later)	Completion routine address.	IRQL value before it calls the completion routine	Current IRQL value, after it calls the completion routine	The IRP completion routine returned at an IRQL that was different from the IRQL the routine was called at.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0xFB (Windows 7 operating systems and later)	Completion routine address	Current thread's APC disable count	The thread's APC disable count before it calls the IRP completion routine	The thread's APC disable count was changed by the driver's IRP completion routine. The APC disable count is decremented each time a driver calls KeEnterCriticalRegion, FsRtlEnterFileSystem, or acquires a mutex. The APC disable count is incremented each time a driver calls KeLeaveCriticalRegion, KeReleaseMutex, or FsRtlExitFileSystem. Because these calls should always be in pairs, the APC disable count should be zero whenever a thread is exited. A negative value indicates that a driver has disabled APC calls without re-enabling them. A positive value indicates that the reverse is true.
0x105 (Windows 7 operating systems and later)	Address of the IRP	The driver uses ExFreePool instead of IoFreeIrp to release the IRP.		
0x10A (Windows 7 operating systems and later)	The driver attempts to charge pool quota to the Idle process.			
0x10B (Windows 7 operating systems and later)	The driver attempts to charge pool quota from a DPC routine. This is incorrect because the current process context is undefined.			
0x110 (Windows 7 operating systems and later)	Address of the Interrupt Service Routine	Address of the extended context that was saved before it executed the ISR	Address of the extended context was saved after it executed the ISR	The interrupt service routine (ISR) for the driver has corrupted the extended thread context.
0x111 (Windows 7 operating systems and later)	Address of the Interrupt Service Routine	IRQL before executing ISR	IRQL after executing ISR	The interrupt Service Routine returned a changed IRQL.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x115 (Windows 7 operating systems and later)	The address of the thread that is responsible for the shutdown, which might be deadlocked	Driver Verifier detected that the system has taken longer than 20 minutes and shutdown is not complete.		
0x11A (Windows 7 operating systems and later)	Current IRQL	The driver calls KeEnterCriticalRegion at IRQL > APC_LEVEL.		
0x11B (Windows 7 operating systems and later)	Current IRQL	The driver calls KeLeaveCriticalRegion at IRQL > APC_LEVEL.		
0x120 (Windows 7 operating systems and later)	Address of the IRQL value	Address of the Object to wait on	Address of Timeout value	The thread waits at IRQL > DISPATCH_LEVEL. Callers of KeWaitForSingleObject or KeWaitForMultipleObjects must run at IRQL <= DISPATCH_LEVEL.
0x121 (Windows 7 operating systems and later)	Address of the IRQL value	Address of the Object to wait on	Address of Timeout value	The thread waits at IRQL equals DISPATCH_LEVEL and the Timeout is NULL. Callers of KeWaitForSingleObject or KeWaitForMultipleObjects can run at IRQL <= DISPATCH_LEVEL. If a NULL pointer is supplied for Timeout, the calling thread remains in a wait state until the Object is signaled.
0x122 (Windows 7 operating systems and later)	Address of the IRQL value	Address of the Object to wait on	Address of the Timeout value	The thread waits at DISPATCH_LEVEL and Timeout value is not equal to zero (0). If the Timeout != 0, the callers of KeWaitForSingleObject or KeWaitForMultipleObjects must run at IRQL <= APC_LEVEL.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x123 (Windows 7 operating systems and later)	Address of the Object to wait on	The caller of KeWaitForSingleObject or KeWaitForMultipleObjects specified the wait as UserMode, but the Object is on the kernel stack.		
0x130 (Windows 7 operating systems and later)	Address of work item	The work item is in session address space. Work items are not allowed in session address space because they can be manipulated from another session or from system threads that have no session virtual address space.		
0x131 (Windows 7 operating systems and later)	Address of work item	The work item is in pageable memory. Work items have to be in nonpageable memory because the kernel uses them at DISPATCH_LEVEL.		
0x135	Address of IRP	Number of milliseconds allowed between the IoCancelIrp call and the completion for this IRP	The canceled IRP did not completed in the expected time The driver took longer than expected to complete the canceled IRP.	
0x13A	Address of the pool block being freed	Incorrect value	Address of the incorrect value	The driver has called ExFreePool and Driver Verifier detects an error in one of the internal values that is used to track pool usage.
0x13B	Address of the pool block being freed	Address of the incorrect value	Address of a pointer to the incorrect memory page	The driver has called ExFreePool and Driver Verifier detects an error in one of the internal values that is used to track pool usage.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x13C	Address of the pool block being freed	Incorrect value	Address of the incorrect value	The driver has called ExFreePool and Driver Verifier detects an error in one of the internal values that is used to track pool usage.
0x13D	Address of the pool block being freed	Address of the incorrect value	Correct value that was expected	The driver has called ExFreePool and Driver Verifier detects an error in one of the internal values that is used to track pool usage.
0x13E	Pool block address specified by the caller	Pool block address tracked by Driver Verifier	Pointer to the pool block address that is tracked by Driver Verifier	The pool block address specified by the caller of ExFreePool is different from the address tracked by Driver Verifier.
0x13F	Address of the pool block being freed	Number of bytes being freed	Pointer to the number of bytes tracked by Driver Verifier	The number of bytes of memory being freed in the call to ExFreePool is different from the number of bytes tracked by Driver Verifier.
0x140	Current IRQL	MDL address	Associated virtual address with this MDL	A non-locked MDL was constructed from either pageable or tradable memory.
0x1000	Address of the resource	Reserved	Reserved	Self-deadlock: The current thread has tried to recursively acquire a resource. A bug check with this parameter occurs only when the Deadlock Detection option of Driver Verifier is active.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x1001	Address of the resource that was the final cause of the deadlock	Reserved	Reserved	Deadlock: A lock hierarchy violation has been found. A bug check with this parameter occurs only when the Deadlock Detection option of Driver Verifier is active. (Use the !deadlock extension for further information.)
0x1002	Address of the resource	Reserved	Reserved	Uninitialized resource: A resource has been acquired without having been initialized first. A bug check with this parameter occurs only when the Deadlock Detection option of Driver Verifier is active.
0x1003	Address of the resource that is being released deadlocked	Address of the resource that should have been released first	Reserved	Unexpected release: A resource has been released in an incorrect order. A bug check with this parameter occurs only when the Deadlock Detection option of Driver Verifier is active.
0x1004	Address of the resource	Address of the thread that acquired the resource	Address of the current thread	Unexpected thread: The wrong thread releases a resource. A bug check with this parameter occurs only when the Deadlock Detection option of Driver Verifier is active.
0x1005	Address of the resource	Reserved	Reserved	Multiple initialization: A resource is initialized more than one time. A bug check with this parameter occurs only when the Deadlock Detection option of Driver Verifier is active.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x1007	Address of the resource	Reserved	Reserved	Unacquired resource: A resource is released before it has been acquired. A bug check with this parameter occurs only when the Deadlock Detection option of Driver Verifier is active.
0x1008 (Windows 7 operating systems and later)	Lock address	Driver Verifier internal data	Driver Verifier internal data	The driver tried to acquire a lock by using an API that is mismatched for this lock type.
0x1009 (Windows 7 operating systems and later)	Lock address	Driver Verifier internal data	Driver Verifier internal data	The driver tried to release a lock by using an API that is mismatched for this lock type.
0x100A (Windows 7 operating systems and later)	Owner thread address	Driver Verifier internal data	The terminated thread owns the lock.	
0x100B (Windows 7 operating systems and later)	Lock address	Owner thread address	Driver Verifier internal address	The deleted lock is still owned by a thread.
0xA001 (Windows 8.1 operating systems and later)	A pointer to the NetBufferList object	A pointer to the virtual switch object (if NON-NULL)	Reserved (unused)	VM Switch: The SourceHandle for the caller-supplied NetBufferList must be set. See the AllocateNetBufferListForwardingContext routine.
0xA002 (Windows 8.1 operating systems and later)	A pointer to the NetBufferList object	A pointer to the virtual switch object (if NON-NULL).	Reserved (unused)	VM Switch: The caller supplied NetBufferList's forwarding detail is not zero. See the AllocateNetBufferListForwardingContext routine.
0xA003 (Windows 8.1 operating systems and later)	A pointer to the NetBufferList object	A pointer to the virtual switch object (if NON-NULL).	Reserved (unused)	VM Switch: The caller supplied a NetBufferList with packet header or routing context that is NULL. See Packet Management Guidelines for the Extensible Switch Data Path.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0xA004 (Windows 8.1 operating systems and later)	ID of invalid port	NIC Index	A pointer to the virtual switch object (if NON-NULL).	VM Switch: The caller specified an invalid Port and NIC index combination. See Hyper-V Extensible Switch Port and Network Adapter States.
0xA005 (Windows 8.1 operating systems and later)	A pointer to the NetBufferList object	A pointer to the Destination list.	A pointer to the virtual switch object (if NON-NULL).	VM Switch: The caller supplied an invalid destination. See AddNetBufferListDestination and UpdateNetBufferListDestinations.
0xA006 (Windows 8.1 operating systems and later)	A pointer to the NetBufferList object	A pointer to the virtual switch object (if NON-NULL).	Reserved (unused)	VM Switch: The caller supplied an invalid source NIC or Port object. See Hyper-V Extensible Switch Port and Network Adapter States.
0xA007 (Windows 8.1 operating systems and later)	A pointer to the NetBufferList object	A pointer to the virtual switch object (if NON-NULL).	Reserved (unused)	VM Switch: The caller supplied an invalid destination list. See AddNetBufferListDestination and UpdateNetBufferListDestinations.
0xA008 (Windows 8.1 operating systems and later)	Parent NIC object	NIC index	A pointer to the virtual switch object (if NON-NULL).	VM Switch: Attempting to reference a NIC when not allowed. See Hyper-V Extensible Switch Port and Network Adapter States.
0xA009 (Windows 8.1 operating systems and later)	Port being referenced	A pointer to the virtual switch object (if NON-NULL)	Reserved (unused)	VM Switch: Attempt to reference a port when not allowed. See Hyper-V Extensible Switch Port and Network Adapter States.
0xA00A (Windows 8.1 operating systems and later)	A pointer to the NetBufferList object	ContextTypeInfo object	Reserved (unused)	VM Switch: Failure context is already set. See SetNetBufferListSwitchContext.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0xA00B (Windows 8.1 operating systems and later)	A pointer to the NetBufferList object	NDIS_SWITCH_REPORT_FILTERED_NBL_FLAGS_*	A pointer to the virtual switch object (if NON-NULL)	VM Switch: Invalid direction provided for dropped NetBufferList. See ReportFilteredNetBufferLists.
0xA00C (Windows 8.1 operating systems and later)	A pointer to the NetBufferList object	Send Flags value	A pointer to the virtual switch object (if NON-NULL)	VM Switch: NetBufferList chain has multiple source ports when NDIS_SEND_FLAGS_SWITCH_SINGLE_SOURCE flag is set. See Hyper-V Extensible Switch Send and Receive Flags.
0xA00D (Windows 8.1 operating systems and later)	A pointer to the NetBufferList object	A pointer to the virtual switch context	A pointer to the virtual switch object (if NON-NULL)	VM Switch: One or more NetBufferLists in chain have invalid destination when NDIS_RECEIVE_FLAGS_SWITCH_DESTINATION_GROUP flag is set. See Hyper-V Extensible Switch Send and Receive Flags.
0x2000 (Windows 7 operating systems and later)	The first argument passed to the StorPortInitialize routine. This parameter is a pointer to the driver object that the operating system passed to the miniport driver in the first argument of the miniport driver's DriverEntry routine.	The second argument passed to the StorPortInitialize routine. This parameter is a pointer to context information that the operating system passed to the miniport driver in the second argument of the miniport driver's DriverEntry routine.	Reserved	The Storport miniport driver passed a bad argument (a NULL pointer) to the StorPortInitialize routine.
0x00020002 (Windows 8 operating systems and later)	Pointer to the string that describes the violated rule condition.	Optional pointer to the rule state variable(s).	Reserved	The driver violated the DDI compliance rule IrqlApcLte. The rule specifies that the driver must call ObGetObjectSecurity and ObReleaseObjectSecurity only when IRQL <= APC_LEVEL.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x00020003 (Windows 8 operating systems and later)	Pointer to the string that describes the violated rule condition.	Optional pointer to the rule state variable(s).	Reserved	The driver violated the DDI compliance rule IrqlDispatch. The IrqlDispatch rule specifies that the driver must call certain routines only when IRQL = DISPATCH_LEVEL
0x00020004 (Windows 8 operating systems and later)	Pointer to the string that describes the violated rule condition.	Optional pointer to the rule state variable(s).	Reserved	The driver violated the DDI compliance rule IrqlExAllocatePool. The IrqlExAllocatePool rule specifies that the driver calls ExAllocatePoolWithTag and ExAllocatePoolWithTagPriority only when at IRQL<=DISPATCH_LEVEL.
0x00020005 (Windows 8 operating systems and later)	Pointer to the string that describes the violated rule condition.	Optional pointer to the rule state variable(s).	Reserved	The driver violated the DDI compliance rule IrqlExApcLte1. The IrqlExApcLte1 rule specifies that the driver calls ExAcquireFastMutex and ExTryToAcquireFastMutex only at IRQL <= APC_LEVEL.
0x00020006 (Windows 8 operating systems and later)	Pointer to the string that describes the violated rule condition.	Optional pointer to the rule state variable(s).	Reserved	The driver violated the DDI compliance rule IrqlExApcLte2. The IrqlExApcLte2 rule specifies that the driver calls certain routines only when IRQL <= APC_LEVEL.
0x00020007 (Windows 8 operating systems and later)	Pointer to the string that describes the violated rule condition.	Optional pointer to the rule state variable(s).	Reserved	The driver violated the DDI compliance rule IrqlExApcLte3. The IrqlExApcLte3 rule specifies that the driver must call certain executive support routines only when IRQL <= APC_LEVEL.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x00020008 (Windows 8 operating systems and later)	Pointer to the string that describes the violated rule condition.	Optional pointer to the rule state variable(s).	Reserved	The driver violated the DDI compliance rule IrqlExPassive. The IrqlExPassive rule specifies that the driver must call certain executive support routines only when IRQL = PASSIVE_LEVEL.
0x00020009 (Windows 8 operating systems and later)	Pointer to the string that describes the violated rule condition.	Optional pointer to the rule state variable(s).	Reserved	The driver violated the DDI compliance rule IrqlIoApcLte. The IrqlIoApcLte rule specifies that the driver must call certain I/O manager routines only when IRQL <= APC_LEVEL.
0x0002000A (Windows 8 operating systems and later)	Pointer to the string that describes the violated rule condition.	Optional pointer to the rule state variable(s).	Reserved	The driver violated the DDI compliance rule IrqlIoPassive1. The IrqlIoPassive1 rule specifies that the driver must call certain I/O manager routines only when IRQL = PASSIVE_LEVEL.
0x0002000B (Windows 8 operating systems and later)	Pointer to the string that describes the violated rule condition.	Optional pointer to the rule state variable(s).	Reserved	The driver violated the DDI compliance rule IrqlIoPassive2. The IrqlIoPassive2 rule specifies that the driver must call certain I/O manager routines only when IRQL = PASSIVE_LEVEL.
0x0002000C (Windows 8 operating systems and later)	Pointer to the string that describes the violated rule condition.	Optional pointer to the rule state variable(s).	Reserved	The driver violated the DDI compliance rule IrqlIoPassive3. The IrqlIoPassive3 rule specifies that the driver must call certain I/O manager routines only when IRQL = PASSIVE_LEVEL.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x0002000D (Windows 8 operating systems and later)	Pointer to the string that describes the violated rule condition.	Optional pointer to the rule state variable(s).	Reserved	The driver violated the DDI compliance rule IrqlIoPassive4. The IrqlIoPassive4 rule specifies that the driver must call certain I/O manager routines only when IRQL = PASSIVE_LEVEL.
0x0002000E (Windows 8 operating systems and later)	Pointer to the string that describes the violated rule condition.	Optional pointer to the rule state variable(s).	Reserved	The driver violated the DDI compliance rule IrqlIoPassive5. The IrqlIoPassive5 rule specifies that the driver must call certain I/O manager routines only when IRQL = PASSIVE_LEVEL.
0x0002000F (Windows 8 operating systems and later)	Pointer to the string that describes the violated rule condition.	Optional pointer to the rule state variable(s).	Reserved	The driver violated the DDI compliance rule IrqlKeApcLte1. The IrqlKeApcLte1 rule specifies that the driver must call certain kernel routines only when IRQL <= APC_LEVEL.
0x00020010 (Windows 8 operating systems and later)	Pointer to the string that describes the violated rule condition.	Optional pointer to the rule state variable(s).	Reserved	The driver violated the DDI compliance rule IrqlKeApcLte2. The IrqlKeApcLte2 rule specifies that the driver must call certain kernel routines only when IRQL <= APC_LEVEL.
0x00020011 (Windows 8 operating systems and later)	Pointer to the string that describes the violated rule condition.	Optional pointer to the rule state variable(s).	Reserved	The driver violated the DDI compliance rule IrqlKeDispatchLte. The IrqlKeDispatchLte rule specifies that the driver must call certain kernel routines only when IRQL <= DISPATCH_LEVEL.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x00020015 (Windows 8 operating systems and later)	Pointer to the string that describes the violated rule condition.	Optional pointer to the rule state variable(s).	Reserved	The driver violated the DDI compliance rule IrqlKeReleaseSpinLock. The IrqlKeReleaseSpinLock rule specifies that the driver must call KeReleaseSpinLock only when IRQL = DISPATCH_LEVEL.
0x00020016 (Windows 8 operating systems and later)	Pointer to the string that describes the violated rule condition.	Optional pointer to the rule state variable(s).	Reserved	The driver violated the DDI compliance rule IrqlKeSetEvent. The IrqlKeSetEvent rule specifies that the KeSetEvent routine is only called at IRQL <= DISPATCH_LEVEL when Wait is set to FALSE, and at IRQL <= APC_LEVEL when Wait is set to TRUE.
0x00020019 (Windows 8 operating systems and later)	Pointer to the string that describes the violated rule condition.	Optional pointer to the rule state variable(s).	Reserved	The driver violated the DDI compliance rule IrqlMmApcLte. The IrqlMmApcLte rule specifies that the driver must call certain memory manager routines only when IRQL <= APC_LEVEL.
0x0002001A (Windows 8 operating systems and later)	Pointer to the string that describes the violated rule condition.	Optional pointer to the rule state variable(s).	Reserved	The driver violated the DDI compliance rule IrqlMmDispatch. The IrqlMmDispatch rule specifies that the driver must call MmFreeContiguousMemory only when IRQL = DISPATCH_LEVEL.
0x0002001B (Windows 8 operating systems and later)	Pointer to the string that describes the violated rule condition.	Optional pointer to the rule state variable(s).	Reserved	The driver violated the DDI compliance rule IrqlObPassive. The IrqlObPassive rule specifies that the driver must call ObReferenceObjectByHandle only when IRQL = PASSIVE_LEVEL.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x0002001C (Windows 8 operating systems and later)	Pointer to the string that describes the violated rule condition.	Optional pointer to the rule state variable(s).	Reserved	The driver violated the DDI compliance rule IrqlPsPassive. The IrqlPsPassive rule specifies that the driver must call certain process and thread manager routines only when IRQL = PASSIVE_LEVEL.
0x0002001D (Windows 8.1 operating systems and later)	Pointer to the string that describes the violated rule condition.	Address of internal rule state (second argument to !ruleinfo).	Address of supplemental states (third argument to !ruleinfo).	The driver violated the DDI compliance rule IrqlReturn.
0x0002001E (Windows 8 operating systems and later)	Pointer to the string that describes the violated rule condition.	Optional pointer to the rule state variable(s).	Reserved	The driver violated the DDI compliance rule IrqlRtlPassive. The IrqlRtlPassive rule specifies that the driver must call RtlDeleteRegistryValue only when IRQL = PASSIVE_LEVEL.
0x0002001F (Windows 8 operating systems and later)	Pointer to the string that describes the violated rule condition.	Optional pointer to the rule state variable(s).	Reserved	The driver violated the DDI compliance rule IrqlZwPassive. The IrqlZwPassive rule specifies that the driver must call ZwClose only when IRQL = PASSIVE_LEVEL.
0x00020022 (Windows 8.1 operating systems and later)	Pointer to the string that describes the violated rule condition.	Reserved (unused)	Reserved (unused)	The driver violated the DDI compliance rule IrqlIoDispatch.
0x00040003 (Windows 8.1 operating systems and later)	Pointer to the string that describes the violated rule condition.	Address of internal rule state (second argument to !ruleinfo).	Address of supplemental states (third argument to !ruleinfo).	The driver violated the DDI compliance rule CriticalRegions.
0x00040006 (Windows 8.1 operating systems and later)	Pointer to the string that describes the violated rule condition.	Address of internal rule state (second argument to !ruleinfo).	Address of supplemental states (third argument to !ruleinfo).	The driver violated the DDI compliance rule QueuedSpinLock.
0x00040007 (Windows 8.1 operating systems and later)	Pointer to the string that describes the violated rule condition.	Address of internal rule state (second argument to !ruleinfo).	Address of supplemental states (third argument to !ruleinfo).	The driver violated the DDI compliance rule QueuedSpinLockRelease.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x00040009 (Windows 8.1 operating systems and later)	Pointer to the string that describes the violated rule condition.	Address of internal rule state (second argument to !ruleinfo).	Address of supplemental states (third argument to !ruleinfo).	The driver violated the DDI compliance rule SpinLock.
0x0004000B (Windows 8.1 operating systems and later)	Pointer to the string that describes the violated rule condition.	Address of internal rule state (second argument to !ruleinfo).	Address of supplemental states (third argument to !ruleinfo).	The driver violated the DDI compliance rule SpinlockRelease.
0x0004000E (Windows 8.1 operating systems and later)	Pointer to the string that describes the violated rule condition.	Address of internal rule state (second argument to !ruleinfo).	Address of supplemental states (third argument to !ruleinfo).	The driver violated the DDI compliance rule GuardedRegions.
0x0004100B (Windows 8.1 operating systems and later)	Pointer to the string that describes the violated rule condition.	Reserved	Reserved	The driver violated the DDI compliance rule RequestedPowerIrp.
0x0004100F (Windows 8.1 operating systems and later)	Pointer to the string that describes the violated rule condition.	Address of internal rule state (second argument to !ruleinfo).	Address of supplemental states (third argument to !ruleinfo).	The driver violated the DDI compliance rule IoSetCompletionExCompleteIrp.
0x00043006 (Windows 8.1 operating systems and later)	Pointer to the string that describes the violated rule condition.	Reserved	Reserved	The driver violated the DDI compliance rule PnpRemove.
0x00091001 (Windows 8.1 operating systems and later)	Pointer to the string that describes the violated rule condition.	Address of internal rule state (second argument to !ruleinfo).	Address of supplemental states (third argument to !ruleinfo).	The driver violated the DDI compliance rule NdisOidComplete.
0x00091002 (Windows 8.1 operating systems and later)	Pointer to the string that describes the violated rule condition.	Address of internal rule state (second argument to !ruleinfo).	Address of supplemental states (third argument to !ruleinfo).	The driver violated the DDI compliance rule NdisOidDoubleComplete.
0x0009100E (Windows 8.1 operating systems and later)	Pointer to the string that describes the violated rule condition.	Address of internal rule state (second argument to !ruleinfo).	Address of supplemental states (third argument to !ruleinfo).	The driver violated the DDI compliance rule NdisOidDoubleRequest.
0x00092003 (Windows 8.1 operating systems and later)	Pointer to the string that describes the violated rule condition.	Address of internal rule state (second argument to !ruleinfo).	Address of supplemental states (third argument to !ruleinfo).	The driver violated the NDIS/WIFI verification rule NdisTimedOidComplete.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x0009200D (Windows 8.1 operating systems and later)	Pointer to the string that describes the violated rule condition.	Address of internal rule state (second argument to !ruleinfo).	Address of supplemental states (third argument to !ruleinfo).	The driver violated the NDIS/WIFI verification rule NdisTimedDataSend.
0x0009200F (Windows 8.1 operating systems and later)	Pointer to the string that describes the violated rule condition.	Address of internal rule state (second argument to !ruleinfo).	Address of supplemental states (third argument to !ruleinfo).	The driver violated the NDIS/WIFI verification rule NdisTimedDataHang.
0x00093004 (Windows 8.1 operating systems and later)	Pointer to the string that describes the violated rule condition.	Address of internal rule state (second argument to !ruleinfo).	Address of supplemental states (third argument to !ruleinfo).	The driver violated the NDIS/WIFI verification rule WlanAssociation.
0x00093005 (Windows 8.1 operating systems and later)	Pointer to the string that describes the violated rule condition.	Address of internal rule state (second argument to !ruleinfo).	Address of supplemental states (third argument to !ruleinfo).	The driver violated the NDIS/WIFI verification rule WlanConnectionRoaming.
0x00093006 (Windows 8.1 operating systems and later)	Pointer to the string that describes the violated rule condition.	Address of internal rule state (second argument to !ruleinfo).	Address of supplemental states (third argument to !ruleinfo).	The driver violated the NDIS/WIFI verification rule WlanDisassociation.
0x00094007 (Windows 8.1 operating systems and later)	Pointer to the string that describes the violated rule condition.	Address of internal rule state (second argument to !ruleinfo).	Address of supplemental states (third argument to !ruleinfo).	The driver violated the NDIS/WIFI verification rule WlanTimedAssociation.
0x00094008 (Windows 8.1 operating systems and later)	Pointer to the string that describes the violated rule condition.	Address of internal rule state (second argument to !ruleinfo).	Address of supplemental states (third argument to !ruleinfo).	The driver violated the NDIS/WIFI verification rule WlanTimedConnectionRoaming.
0x00094009 (Windows 8.1 operating systems and later)	Pointer to the string that describes the violated rule condition.	Address of internal rule state (second argument to !ruleinfo).	Address of supplemental states (third argument to !ruleinfo).	The driver violated the NDIS/WIFI verification rule WlanTimedConnectRequest.
0x0009400B (Windows 8.1 operating systems and later)	Pointer to the string that describes the violated rule condition.	Address of internal rule state (second argument to !ruleinfo).	Address of supplemental states (third argument to !ruleinfo).	The driver violated the NDIS/WIFI verification rule WlanTimedLinkQuality.
0x0009400C (Windows 8.1 operating systems and later)	Pointer to the string that describes the violated rule condition.	Address of internal rule state (second argument to !ruleinfo).	Address of supplemental states (third argument to !ruleinfo).	The driver violated the NDIS/WIFI verification rule WlanTimedScan.

Cause

See the description of each code in the Parameters section for a description of the cause. Further information can be obtained by using the **!analyze -v** extension.

Resolution

This bug check can only occur when Driver Verifier has been instructed to monitor one or more drivers. If you did not intend to use Driver Verifier, you should deactivate it. You might also consider removing the driver that caused this problem.

If you are the driver writer, use the information obtained through this bug check to fix the bugs in your code.

For full details on Driver Verifier, see the Driver Verifier section of the Windows Driver Kit (WDK).

Remarks

The _POOL_TYPE codes are enumerated in Ntddk.h. In particular, **0** (zero) indicates nonpaged pool and **1** (one) indicates paged pool.

(Windows 8 and later versions of Windows) If DDI compliance checking causes a bug check, run Static Driver Verifier on the driver source code and specify the DDI compliance rule (identified by the parameter 1 value) that caused the bug check. Static Driver Verifier can help you locate the cause of the problem in your source code.

See also

Handling a Bug Check When Driver Verifier is Enabled

Bug Check 0xC5: DRIVER_CORRUPTED_EXPOOL

7/2/2019 • 2 minutes to read • Edit Online

The DRIVER_CORRUPTED_EXPOOL bug check has a value of 0x000000C5. This indicates that the system attempted to access invalid memory at a process IRQL that was too high.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

DRIVER_CORRUPTED_EXPOOL Parameters

PARAMETER	DESCRIPTION
1	Memory referenced
2	IRQL at time of reference
3	**0:** Read **1:** Write
4	Address that referenced memory

Cause

The kernel attempted to access pageable memory (or perhaps completely invalid memory) when the IRQL was too high. The ultimate cause of this problem is almost certainly a driver that has corrupted the system pool.

In most cases, this bug check results if a driver corrupts a small allocation (less than PAGE_SIZE). Larger allocations result in **bug check 0xD0** (DRIVER_CORRUPTED_MMPOOL).

Resolution

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause. If you have recently installed any new software, check to see if it is properly installed. Check for updated drivers on the manufacturer's website.

To debug this error, use the special pool option of Driver Verifier. If this fails to reveal the driver that caused the error, use the Global Flags utility to enable the special pool by pool tag.

For information about the special pool, consult the Driver Verifier section of the Windows Driver Kit.

Bug Check 0xC6: DRIVER_CAUGHT_MODIFYING_FREED_POOL

7/2/2019 • 2 minutes to read • Edit Online

The DRIVER_CAUGHT_MODIFYING_FREED_POOL bug check has a value of 0x000000C6. This indicates that the driver attempted to access a freed memory pool.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

DRIVER_CAUGHT_MODIFYING_FREED_POOL Parameters

PARAMETER	DESCRIPTION
1	Memory referenced
2	**0:** Read **1:** Write
3	**0:** Kernel mode **1:** User mode
4	Reserved

Remarks

The faulty component will be displayed in the current kernel stack. This driver should be either replaced or debugged.

Bug Check 0xC7: TIMER_OR_DPC_INVALID

7/2/2019 • 2 minutes to read • Edit Online

The TIMER_OR_DPC_INVALID bug check has a value of 0x000000C7. This is issued if a kernel timer or delayed procedure call (DPC) is found somewhere in memory where it is not permitted.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

TIMER_OR_DPC_INVALID Parameters

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x0	Address of the timer object	Start of memory range being checked	End of memory range being checked	The timer object was found in a block of memory where a timer object is not permitted. .
0x1	Address of the DPC object	Start of memory range being checked	End of memory range being checked	The DPC object was found in a block of memory where a DPC object is not permitted.
0x2	Address of the DPC routine	Start of memory range being checked	End of memory range being checked	The DPC routine was found in a block of memory where a DPC object is not permitted.
0x3	Address of the DPC object	Processor number	Number of processors in the system	The processor number for the DPC object is not correct.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x4	Address of the DPC routine	The thread's APC disable count before the kernel calls the DPC routine	The thread's APC disable count after the DPC routine is called	The thread's APC disable count was changed during DPC routine execution. The APC disable count is decremented each time a driver calls **KeEnterCriticalRegion**, **FsRtlEnterFileSystem**, or acquires a mutex. The APC disable count is incremented each time a driver calls **KeLeaveCriticalRegion**, **KeReleaseMutex**, or **FsRtlExitFileSystem**.
0x5	Address of the DPC routine	The thread's APC disable count before the kernel calls the DPC routine	The thread's APC disable count after the DPC routine is called	The thread's APC disable count was changed during the execution of timer DPC routine. The APC disable count is decremented each time a driver calls **KeEnterCriticalRegion**, **FsRtlEnterFileSystem**, or acquires a mutex. The APC disable count is incremented each time a driver calls **KeLeaveCriticalRegion**, **KeReleaseMutex**, or **FsRtlExitFileSystem**.

Cause

This condition is usually caused by a driver failing to cancel a timer or DPC before freeing the memory where it resides.

Resolution

If you are the driver writer, use the information obtained through this bug check to fix the bugs in your code.

If you are a system administrator, you should unload the driver if the problem persists.

Bug Check 0xC8: IRQL_UNEXPECTED_VALUE

7/2/2019 • 2 minutes to read • Edit Online

The IRQL_UNEXPECTED_VALUE bug check has a value of 0x000000C8. This indicates that the processor's IRQL is not what it should be at this time.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

IRQL_UNEXPECTED_VALUE Parameters

PARAMETER	DESCRIPTION
1	The value of the following bit computation: (Current IRQL << 16) | (Expected IRQL << 8) | UniqueValue
2	Zero, or **APC->KernelRoutine**
3	Zero, or **APC**
4	Zero, or **APC->NormalRoutine**

You can determine "UniqueValue" by computing (Parameter 1 AND 0xFF). If "UniqueValue" is either zero or one, Parameter 2, Parameter 3, and Parameter 4 will equal the indicated APC pointers. Otherwise, these parameters will equal zero.

Cause

This error is usually caused by a device driver or another lower-level program that changed the IRQL for some period and did not restore the original IRQL at the end of that period. For example, the routine may have acquired a spin lock and failed to release it.

Bug Check 0xC9: DRIVER_VERIFIER_IOMANAGER_VIOLATION

7/2/2019 • 22 minutes to read • Edit Online

The DRIVER_VERIFIER_IOMANAGER_VIOLATION bug check has a value of 0x000000C9. This is the bug check code for all Driver Verifier **I/O Verification** violations.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

DRIVER_VERIFIER_IOMANAGER_VIOLATION Parameters

When Driver Verifier is active and **I/O Verification** is selected, various I/O violations will cause this bug check to be issued. Parameter 1 identifies the type of violation.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x01	Address of IRP being freed	0	0	The driver attempted to free an object whose type is not IO_TYPE_IRP.
0x02	Address of IRP being freed	0	0	The driver attempted to free an IRP that is still associated with a thread.
0x03	Address of IRP being sent	0	0	The driver passed **IoCallDriver** an IRP Type not equal to IRP_TYPE.
0x04	Address of device object	0	0	The driver passed **IoCallDriver** an invalid device object.
0x05	Address of device object associated with offending driver	IRQL before **IoCallDriver**	IRQL after **IoCallDriver**	The IRQL changed during a call to the driver dispatch routine.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x06	IRP status	Address of IRP being completed	0	The driver called **IoCompleteRequest** with a status marked as pending (or equal to -1).
0x07	Address of cancel routine	Address of IRP being completed	0	The driver called **IoCompleteRequest** while its cancel routine was still set.
0x08	Address of device object	IRP major function code	Exception status code	The driver passed **IoBuildAsynchronousFsdRequest** an invalid buffer.
0x09	Address of device object	I/O control code	Exception status code	The driver passed **IoBuildDeviceIoControlRequest** an invalid buffer.
0x10	Current IRQL	Reserved	Reserved	IoCallDriver was called above DISPATCH_LEVEL.
0x11	Driver fast I/O dispatch routine address	IRQL before calling driver dispatch routine	Current IRQL	IoCallDriver was called above DISPATCH_LEVEL.
0x12	Driver dispatch routine address	IRQL before calling driver dispatch routine	Current IRQL	IoCallDriver was called above DISPATCH_LEVEL.
0x0A	Address of device object	0	0	The driver passed **IoInitializeTimer** a device object with an already-initialized timer.
0x0C	Address of I/O status block	0	0	The driver passed an I/O status block to an IRP, but this block is allocated on a stack which has already unwound past that point.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x0D	Address of user event object	0	0	The driver passed a user event to an IRP, but this event is allocated on a stack which has already unwound past that point.
0x0E	Current IRQL	Address of IRP	0	The driver called **IoCompleteRequest** with IRQL > DISPATCH_LEVEL.
0x0F	Address of the device object to which the IRP is being sent	Pointer to the IRP	Pointer to file object	The driver sent a create request with a file object that has been closed, or that had its open canceled.

In addition to the errors mentioned in the previous table, there are a number of **I/O Verification** errors that will cause Driver Verifier to halt the system, but which are not actually bug checks.

These errors cause messages to be displayed on the blue screen, in a crash dump file, and in a kernel debugger. These messages will appear differently in each of these locations. When these errors occur, the hexadecimal bug check code 0xC9 and the bug check string DRIVER_VERIFIER_IOMANAGER_VIOLATION do not appear on the blue screen or in the debugger, although they will appear in a crash dump file.

On the blue screen, the following data will be displayed:

- The message **IO SYSTEM VERIFICATION ERROR**.
- The message **WDM DRIVER ERROR** *XXX*, where *XXX* is a hexadecimal code representing the specific error. (See the table below for a list of the I/O error codes and their meanings.)
- The name of the driver which caused the error.
- Typically, the address in the driver's code where the error was detected (Parameter 2).

If a kernel-mode crash dump has been enabled, the following information will appear in the crash dump file:

- The message **BugCheck 0xC9 (DRIVER_VERIFIER_IOMANAGER_VIOLATION)**.
- The hexadecimal I/O error code. (See the table below for a list of the I/O error codes and their meanings.)
- Typically, the address in the driver's code where the error was detected (Parameter 2).

If a kernel debugger is attached to the system which has caused this violation, the following information will be sent to the debugger:

- The message **WDM DRIVER ERROR**, along with an assessment of the severity of the error.
- The name of the driver which caused the error.

- A descriptive string which explains the cause of this error. Often additional information is passed along, such as a pointer to the IRP or a pointer to the device object or IRQL information. (See the table below for a list of these descriptive strings and what additional information is specified.)
- A query for further action. Possible responses are **b** (break), **i** (ignore), **z** (zap), **r** (remove), or **d** (disable). Instructing the operating system to continue allows you to see what would happen "down the line" if this error had not occurred. Of course, this often will lead to additional bug checks. The "zap" option will actually remove the breakpoint that caused this error to be discovered.

Note No other bug checks can be ignored in this manner. Only this kind of **I/O Verification** errors can be ignored, and even these errors can only be ignored if a kernel debugger is attached.

The following table lists those **I/O Verification** errors that can appear.

I/O ERROR CODE	SEVERITY	CAUSE OF ERROR
0x200	Unknown	This code covers all unknown **I/O Verification** errors.
0x201	Fatal error	A device is deleting itself while there is another device beneath it in the driver stack. This may be because the caller has forgotten to call **IoDetachDevice** first, or the lower driver may have incorrectly deleted itself. Param 2 - The address in the driver's code where the error was detected.
0x202	Fatal error	A driver has attempted to detach from a device object that is not attached to anything. This may occur if detach was called twice on the same device object. Param 2 - The address in the driver's code where the error was detected. Param 3 - Device object address.
0x203	Fatal error	A driver has called **IoCallDriver** without setting the cancel routine in the IRP to **NULL**. Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address.
0x204	Fatal error	The caller has passed in **NULL** as a device object. This is fatal. Param 2 - Reserved Param 3 - Reserved

I/O ERROR CODE	SEVERITY	CAUSE OF ERROR
0x205	Fatal error	The caller is forwarding an IRP that is currently queued beneath it. The code handling IRPs returning STATUS_PENDING in this driver appears to be broken. Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address.
0x206	Fatal error	The caller has incorrectly forwarded an IRP (control field not zeroed). The driver should use **IoCopyCurrentIrpStackLocationToNext** or **IoSkipCurrentIrpStackLocation**. Param 2 - Reserved Param 3 - Reserved
0x207	Fatal error	The caller has manually copied the stack and has inadvertently copied the upper layer's completion routine. The driver should use **IoCopyCurrentIrpStackLocationToNext**. Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address.
0x208	Fatal error	This IRP is about to run out of stack locations. Someone may have forwarded this IRP from another stack. Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address.
0x209	Fatal error	The caller is completing an IRP that is currently queued beneath it. The code handling IRPs returning STATUS_PENDING in this driver appears to be broken. Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address.

I/O ERROR CODE	SEVERITY	CAUSE OF ERROR
0x20A	Fatal error	The caller of **IoFreeIrp** is freeing an IRP that is still in use. Param 2 - Reserved Param 3 - Reserved
0x20B	Fatal error	The caller of **IoFreeIrp** is freeing an IRP that is still in use. Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address.
0x20C	Fatal error	The caller of **IoFreeIrp** is freeing an IRP that is still queued against a thread. Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address.
0x20D	Fatal error	The caller of **IoInitializeIrp** has passed an IRP that was allocated with **IoAllocateIrp**. This is illegal and unnecessary, and has caused a quota leak. Check the documentation for **IoReuseIrp** if this IRP is being recycled. Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address.
0x20E	Non-fatal error	A PNP IRP has an invalid status. (Any PNP IRP must have its status initialized to STATUS_NOT_SUPPORTED.) Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address.

I/O ERROR CODE	SEVERITY	CAUSE OF ERROR
0x20F	Non-fatal error	A Power IRP has an invalid status. (Any Power IRP must have its status initialized to STATUS_NOT_SUPPORTED.) Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address.
0x210	Non-fatal error	A WMI IRP has an invalid status. (Any WMI IRP must have its status initialized to STATUS_NOT_SUPPORTED.) Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address.
0x211	Non-fatal error	The caller has forwarded an IRP while skipping a device object in the stack. The caller is probably sending IRPs to the PDO instead of to the device returned by **IoAttachDeviceToDeviceStack**. Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address.
0x212	Non-fatal error	The caller has trashed or has not properly copied the IRP's stack. Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address.
0x213	Non-fatal error	The caller has changed the status field of an IRP it does not understand. Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address.

I/O ERROR CODE	SEVERITY	CAUSE OF ERROR
0x214	Non-fatal error	The caller has changed the information field of an IRP it does not understand. Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address.
0x215	Non-fatal error	A non-successful non-STATUS_NOT_SUPPORTED IRP status for IRP_MJ_PNP is being passed down stack. Failed PNP IRPs must be completed. Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address.
0x216	Non-fatal error	The previously-set IRP_MJ_PNP status has been converted to STATUS_NOT_SUPPORTED. This failure status is reserved for use by the operating system. Drivers cannot fail a PnP IRP with this value. Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address.
0x217	Non-fatal error	The driver has not handled a required IRP. The driver must update the status of the IRP to indicate whether or not it has been handled. Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address.
0x218	Non-fatal error	The driver has responded to an IRP that is reserved for other device objects elsewhere in the stack. Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address.

I/O ERROR CODE	SEVERITY	CAUSE OF ERROR
0x219	Non-fatal error	A non-successful non-STATUS_NOT_SUPPORTED IRP status for IRP_MJ_POWER is being passed down stack. Failed POWER IRPs must be completed. Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address.
0x21A	Non-fatal error	The previously-set IRP_MJ_POWER status has been converted to STATUS_NOT_SUPPORTED. Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address.
0x21B	Non-fatal error	A driver has returned a suspicious status. This is probably due to an uninitialized variable bug in the driver. Param 3 - IRP address.
0x21C	Warning	The caller has copied the IRP stack but not set a completion routine. This is inefficient -- use **IoSkipCurrentIrpStackLocation** instead. Param 2 - Reserved Param 3 - Reserved
0x21D	Fatal error	An IRP dispatch handler has not properly detached from the stack below it upon receiving a remove IRP. Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address. Param 4 - Device object address.

I/O ERROR CODE	SEVERITY	CAUSE OF ERROR
0x21E	Fatal error	An IRP dispatch handler has not properly deleted its device object upon receiving a remove IRP. Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address. Param 4 - Device object address.
0x21F	Non-fatal error	A driver has not filled out a dispatch routine for a required IRP major function. Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address.
0x220	Non-fatal error	IRP_MJ_SYSTEM_CONTROL has been completed by someone other than the ProviderId. This IRP should either have been completed earlier or should have been passed down. Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address. Param 4 - ProviderId.
0x221	Fatal error	An IRP dispatch handler for a PDO has deleted its device object, but the hardware has not been reported as missing in a bus relations query. Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address. Param 4 - Device object address.

I/O ERROR CODE	SEVERITY	CAUSE OF ERROR
0x222	Fatal error	A Bus Filter's IRP dispatch handler has detached upon receiving a remove IRP when the PDO is still alive. Bus Filters must clean up in **FastIoDetach** callbacks. Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address. Param 4 - Device object address.
0x223	Fatal error	An IRP dispatch handler for a bus filter has deleted its device object, but the PDO is still present. Bus filters must clean up in **FastIoDetach** callbacks. Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address. Param 4 - Device object address.
0x224	Fatal error	An IRP dispatch handler has returned a status that is inconsistent with the IRP's **IoStatus.Status** field. Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address. Param 4 - Expected status code.
0x225	Non-fatal error	An IRP dispatch handler has returned a status that is illegal (0xFFFFFFFF). This is probably due to an uninitialized stack variable. To debug this error, use the **ln (List Nearest Symbols)** command with the specified address. Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address. Param 4 - Status code.

I/O ERROR CODE	SEVERITY	CAUSE OF ERROR
0x226	Fatal error	An IRP dispatch handler has returned without passing down or completing this IRP, or someone forgot to return STATUS_PENDING. Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address.
0x227	Fatal error	An IRP completion routine is in pageable code. (This is never permitted.) Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address.
0x228	Non-fatal error	A driver's completion routine has not marked the IRP pending if the **PendingReturned** field was set in the IRP passed to it. This may cause Windows to hang, especially if an error is returned by the stack. Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address.
0x229	Fatal error	A cancel routine has been set for an IRP that is currently being processed by drivers lower in the stack, possibly stomping their cancel routine. Param 2 - Reserved. Param 3 - Reserved.
0x22A	Non-fatal error	The physical device object (PDO) has not responded to a required IRP. Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address.

I/O ERROR CODE	SEVERITY	CAUSE OF ERROR
0x22B	Non-fatal error	The physical device object (PDO) has forgotten to fill out the device relation list with the PDO for the **TargetDeviceRelation** query. Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address.
0x22C	Fatal error	The code implementing the **TargetDeviceRelation** query has not called **ObReferenceObject** on the PDO. Param 2 - Reserved. Param 3 - Reserved.
0x22D	Non-fatal error	The caller has completed a IRP_MJ_PNP it didn't understand instead of passing it down. Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address.
0x22E	Non-fatal error	The caller has completed a successful IRP_MJ_PNP instead of passing it down. Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address.
0x22F	Non-fatal error	The caller has completed an untouched IRP_MJ_PNP (instead of passing the IRP down), or non-PDO has failed the IRP using illegal value of STATUS_NOT_SUPPORTED. Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address.

I/O ERROR CODE	SEVERITY	CAUSE OF ERROR
0x230	Non-fatal error	The caller has completed an IRP_MJ_POWER it didn't understand instead of passing it down. Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address.
0x231	Fatal error	The caller has completed a successful IRP_MJ_POWER instead of passing it down. Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address.
0x232	Non-fatal error	The caller has completed an untouched IRP_MJ_POWER (instead of passing the IRP down), or non-PDO has failed the IRP using illegal value of STATUS_NOT_SUPPORTED. Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address.
0x233	Non-fatal error	The version field of the query capabilities structure in a query capabilities IRP was not properly initialized. Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address.
0x234	Non-fatal error	The size field of the query capabilities structure in a query capabilities IRP was not properly initialized. Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address.

I/O ERROR CODE	SEVERITY	CAUSE OF ERROR
0x235	Non-fatal error	The address field of the query capabilities structure in a query capabilities IRP was not properly initialized to -1. Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address.
0x236	Non-fatal error	The UI Number field of the query capabilities structure in a query capabilities IRP was not properly initialized to -1. Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address.
0x237	Fatal error	A driver has sent an IRP that is restricted for system use only. Param 2 - Reserved. Param 3 - Reserved.
0x238	Warning	The caller of **IoInitializeIrp** has passed an IRP that was allocated with **IoAllocateIrp**. This is illegal, unnecessary, and negatively impacts performance in normal use. If this IRP is being recycled, see **IoReuseIrp** in the Windows Driver Kit. Param 2 - Reserved. Param 3 - Reserved.
0x239	Warning	The caller of **IoCompleteRequest** is completing an IRP that has never been forwarded via a call to **IoCallDriver** or **PoCallDriver**. This may be a bug. Param 2 - Reserved. Param 3 - Reserved.

I/O ERROR CODE	SEVERITY	CAUSE OF ERROR
0x23A	Fatal error	A driver has forwarded an IRP at an IRQL that is illegal for this major code. Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address.
0x23B	Non-fatal error	The caller has changed the status field of an IRP it does not understand. Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address.

The following table lists additional **I/O Verification** errors that can appear. These errors appear when **Enhanced I/O Verification** is activated. For more information, see Enhanced I/O Verification.

I/O ERROR CODE	SEVERITY	CAUSE OF ERROR
0x23C	Fatal error	A driver has completed an IRP without setting the cancel routine in the IRP to **NULL**. Param 2 - Reserved. Param 3 - Reserved.
0x23D	Non-fatal error	A driver has returned STATUS_PENDING but did not mark the IRP pending via a call to **IoMarkIrpPending**. Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address. Param 4 - Status code.
0x23E	Non-fatal error	A driver has marked an IRP pending but didn't return STATUS_PENDING. Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address. Param 4 - Status code.

I/O ERROR CODE	SEVERITY	CAUSE OF ERROR
0x23F	Fatal error	A driver has not inherited the DO_POWER_PAGABLE bit from the stack it has attached to. Param 2 - Reserved. Param 3 - Reserved.
0x240	Fatal error	A driver is attempting to delete a device object that has already been deleted via a prior call to **IoDeleteDevice**. Param 2 - The address in the driver's code where the error was detected. Param 2 - Reserved. Param 3 - Reserved.
0x241	Fatal error	A driver has detached its device object during a surprise remove IRP. Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address. Param 4 - Device object address.
0x242	Fatal error	A driver has deleted its device object during a surprise remove IRP. Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address. Param 4 - Device object address.
0x243	Fatal error	A driver has failed to clear the DO_DEVICE_INITIALIZING flag at the end of **AddDevice**. Param 2 - Reserved. Param 3 - Reserved. Param 4 -

I/O ERROR CODE	SEVERITY	CAUSE OF ERROR
0x244	Fatal error	A driver has not copied either the DO_BUFFERED_IO or the DO_DIRECT_IO flag from the device object it is attaching to. Param 2 - Reserved. Param 3 - Reserved.
0x245	Fatal error	A driver has set both the DO_BUFFERED_IO and the DO_DIRECT_IO flags. These flags are mutually exclusive. Param 2 - Reserved. Param 3 - Reserved.
0x246	Fatal error	A driver has failed to copy the **DeviceType** field from the device object it is attaching to. Param 2 - Reserved. Param 3 - Reserved.
0x247	Fatal error Param 2 - Reserved. Param 3 - Reserved.	A driver has failed an IRP that cannot legally be failed.
0x248	Fatal error	A driver has added a device object that is not a PDO to a device relations query. Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address. Param 4 - Device object address.
0x249	Non-fatal error	A driver has enumerated two child PDOs that returned identical Device IDs. Param 2 - The address in the driver's code where the error was detected. Param 3 - First device object address. Param 4 - Second device object address.

I/O ERROR CODE	SEVERITY	CAUSE OF ERROR
0x24A	Fatal error	A driver has mistakenly called a file I/O function with IRQL not equal to PASSIVE_LEVEL. Param 2 - The address in the driver's code where the error was detected. Param 3 - Reserved. Param 4 - Reserved.
0x24B	Fatal error	A driver has completed an IRP_MN_QUERY_DEVICE_RELATIONS request of type **TargetDeviceRelation** as successful, but did not properly fill out the request or forward the IRP to the underlying hardware stack. Param 2 - The address in the driver's code where the error was detected. Param 3 - Device object address.
0x24C	Non-fatal error	A driver has returned STATUS_PENDING but did not mark the IRP pending by a call to **IoMarkIrpPending**. Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address. Param 4 - Status code.
0x24D	Fatal error	A driver has passed an invalid device object to a function that requires a PDO. Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address. Param 4 - Device object address.

I/O ERROR CODE	SEVERITY	CAUSE OF ERROR
0x300	Non-fatal error	A driver has returned a suspicious status. This is probably due to an uninitialized variable bug in the driver. Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address. Param 4 - Suspicious status code.
0x301	Non-fatal error	A driver has forwarded an IRP at IRQL > DISPATCH_LEVEL. Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address. Param 4 - Incorrect IRQL value
0x302	Non-fatal error	A driver has forwarded an IRP at IRQL > = APC_LEVEL. The I/O Manager will need to queue an APC to complete this request. The APC will not be able to run because the caller is already at APC level, so the caller is likely to deadlock. Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address. Param 4 - Incorrect IRQL value.
0x306	Non-fatal error	The driver is completing an IRP_MJ_PNP (major) and IRP_MN_REMOVE_DEVICE (minor) request with a failure status code. Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address.

I/O ERROR CODE	SEVERITY	CAUSE OF ERROR
0x307	Non-fatal error	The driver issued an I/O request with an event that was already signaled and received a STATUS_PENDING response. This can result in unwinding before the I/O is complete. Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address.
0x310	Non-fatal error	The driver is reinitializing an IRP that is still in use. Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address.
0x311	Non-fatal error	The driver is reinitializing an IRP that was created with IoMakeAssociatedIrp, IoBuildAsynchronousFsdRequest, IoBuildSynchronousFsdRequest, IoBuildDeviceIoControlRequest. Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address.
0x312	Non-fatal error	The caller provided the IRP Status Information field with a value that is greater than the output section of the system buffer. Param 2 - The address in the driver's code where the error was detected. Param 3 - IRP address.

Cause

See the description of each code in the Parameters section for a description of the cause.

Resolution

This bug check can only occur when Driver Verifier has been instructed to monitor one or more drivers. If you did not intend to use Driver Verifier, you should deactivate it. For more information, see "How to control Driver Verifier" in Driver Verifier. You might consider updating or removing the driver which caused this problem as well.

If you are the driver developer, use the information obtained through this bug check to fix the bugs in your code.

For full details on Driver Verifier, see Driver Verifier.

Bug Check 0xCA: PNP_DETECTED_FATAL_ERROR

7/2/2019 • 2 minutes to read • Edit Online

The PNP_DETECTED_FATAL_ERROR bug check has a value of 0x000000CA. This indicates that the Plug and Play Manager encountered a severe error, probably as a result of a problematic Plug and Play driver.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

PNP_DETECTED_FATAL_ERROR Parameters

Parameter 1 identifies the type of violation.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x1	Address of newly-reported PDO	Address of older PDO which has been duplicated	Reserved	**Duplicate PDO:** A specific instance of a driver has enumerated multiple PDOs with identical device ID and unique IDs.
0x2	Address of purported PDO	Address of driver object	Reserved	**Invalid PDO:** An API which requires a PDO has been called with random memory, or with an FDO, or with a PDO which hasn't been initialized. (An uninitialized PDO is one that has not been returned to Plug and Play by **QueryDeviceRelation** or **QueryBusRelations**.)

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x3	Address of PDO whose IDs were queried	Address of ID buffer	**1:** DeviceID **2:** UniqueID **3:** HardwareIDs **4:** CompatibleIDs	**Invalid ID:** An enumerator has returned an ID which contains illegal characters or isn't properly terminated. (IDs must contain only characters in the ranges 0x20 - 0x2B and 0x2D - 0x7F.)
0x4	Address of PDO with DOE_DELETE_PENDING set	Reserved	Reserved	**Invalid enumeration of deleted PDO:** An enumerator has returned a PDO which it had previously deleted using **IoDeleteDevice**.
0x5	Address of PDO	Reserved	Reserved	**PDO freed while linked in devnode tree:** The object manager reference count on a PDO dropped to zero while the devnode was still linked in the tree. (This usually indicates that the driver is not adding a reference when returning the PDO in a query IRP.)
0x8	Address of PDO whose stack returned the invalid bus relation	Total number of PDOs returned as bus relations	The index (zero-based) at which the first **NULL** PDO was found	**NULL pointer returned as a bus relation:** One or more of the devices present on the bus is a **NULL** PDO.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x9	Connection type that was passed	Reserved	Reserved	**Invalid connection type passed to IoDisconnectInterruptEx:** A driver has passed an invalid connection type to **IoDisconnectInterruptEx**. The connection type passed to this routine must match the one returned by a corresponding successful call to **IoConnectInterruptEx**.
0xA	Driver object	IRQL after returning from driver callback	Combined APC disable count after returning from driver callback	**Incorrect notify callback behavior:** A driver failed to preserve IRQL or combined APC disable count across a Plug 'n' Play notification.
0xB	Related PDO	Removal relations	Reserved	**Deleted PDO reported as relation:** One of the removal relations for the device being removed has already been deleted.

Bug Check 0xCB: DRIVER_LEFT_LOCKED_PAGES_IN_PROCESS

7/2/2019 • 2 minutes to read • Edit Online

The DRIVER_LEFT_LOCKED_PAGES_IN_PROCESS bug check has a value of 0x000000CB. This indicates that a driver or the I/O manager failed to release locked pages after an I/O operation.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

DRIVER_LEFT_LOCKED_PAGES_IN_PROCESS Parameters

The four parameters listed in the message can have two possible meanings.

If a driver locked these pages, the parameters have the following meaning.

PARAMETER	DESCRIPTION
1	Calling address in the driver that locked the pages
2	Caller of the calling address in driver that locked the pages
3	Address of the MDL containing the locked pages
4	Number of locked pages

If the driver responsible for the error can be identified, its name is printed on the blue screen and stored in memory at the location (PUNICODE_STRING) **KiBugCheckDriver**.

If the I/O manager locked these pages, the parameters have the following meaning.

PARAMETER	DESCRIPTION
1	Address of the dispatch routine of the top driver on the stack to which the IRP was sent
2	Address of the device object of the top driver on the stack to which the IRP was sent
3	Address of the MDL containing the locked pages

PARAMETER	DESCRIPTION
4	Number of locked pages

Remarks

This bug check is issued only if the registry value **\\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\TrackLockedPages** is equal to DWORD 1. If this value is not set, the system will issue the less-informative **bug check 0x76** (PROCESS_HAS_LOCKED_PAGES).

Starting with Windows Vista, this bug check can also be issued by Driver Verifier when the Pool Tracking option is enabled.

Bug Check 0xCC: PAGE_FAULT_IN_FREED_SPECIAL_POOL

7/2/2019 • 2 minutes to read • Edit Online

The PAGE_FAULT_IN_FREED_SPECIAL_POOL bug check has a value of 0x000000CC. This indicates that the system has referenced memory which was earlier freed.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

PAGE_FAULT_IN_FREED_SPECIAL_POOL Parameters

PARAMETER	DESCRIPTION
1	Memory address referenced
2	**0:** Read **1:** Write
3	Address that referenced memory (if known)
4	Reserved

If the driver responsible for the error can be identified, its name is printed on the blue screen and stored in memory at the location (PUNICODE_STRING) **KiBugCheckDriver**.

Cause

The Driver Verifier Special Pool option has caught the system accessing memory which was earlier freed. This usually indicates a system-driver synchronization problem.

For information about the special pool, consult the Driver Verifier section of the Windows Driver Kit.

Remarks

This cannot be protected by a **try - except** handler -- it can only be protected by a probe.

Bug Check 0xCD: PAGE_FAULT_BEYOND_END_OF_ALLOCATION

7/2/2019 • 2 minutes to read • Edit Online

The PAGE_FAULT_BEYOND_END_OF_ALLOCATION bug check has a value of 0x000000CD. This indicates that the system accessed memory beyond the end of some driver's pool allocation.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

PAGE_FAULT_BEYOND_END_OF_ALLOCATION Parameters

PARAMETER	DESCRIPTION
1	Memory address referenced
2	**0:** Read **1:** Write
3	Address that referenced memory (if known)
4	Reserved

If the driver responsible for the error can be identified, its name is printed on the blue screen and stored in memory at the location (PUNICODE_STRING) **KiBugCheckDriver**.

Cause

The driver allocated *n* bytes of memory from the special pool. Subsequently, the system referenced more than *n* bytes from this pool. This usually indicates a system-driver synchronization problem.

For information about the special pool, consult the Driver Verifier section of the Windows Driver Kit.

Remarks

This cannot be protected by a **try - except** handler -- it can only be protected by a probe.

Bug Check 0xCE: DRIVER_UNLOADED_WITHOUT_CANCELLING_PENDING_OPERATIONS

7/2/2019 • 2 minutes to read • Edit Online

The DRIVER_UNLOADED_WITHOUT_CANCELLING_PENDING_OPERATIONS bug check has a value of 0x000000CE. This indicates that a driver failed to cancel pending operations before unloading.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

DRIVER_UNLOADED_WITHOUT_CANCELLING_PENDING_OPERATIONS Parameters

PARAMETER	DESCRIPTION
1	Memory address referenced
2	**0:** Read **1:** Write
3	Address that referenced memory (if known)
4	Reserved

If the driver responsible for the error can be identified, its name is printed on the blue screen and stored in memory at the location (PUNICODE_STRING) **KiBugCheckDriver**.

Cause

This driver failed to cancel lookaside lists, DPCs, worker threads, or other such items before unload.

Bug Check 0xCF: TERMINAL_SERVER_DRIVER_MADE_INCORRECT_MEMORY_REFERENCE

7/2/2019 • 2 minutes to read • Edit Online

The TERMINAL_SERVER_DRIVER_MADE_INCORRECT_MEMORY_REFERENCE bug check has a value of 0x000000CF. This indicates that a driver has been incorrectly ported to the terminal server.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

TERMINAL_SERVER_DRIVER_MADE_INCORRECT_MEMORY_REFERENCE Parameters

PARAMETER	DESCRIPTION
1	Memory address referenced
2	**0:** Read **1:** Write
3	Address that referenced memory (if known)
4	Reserved

If the driver responsible for the error can be identified, its name is printed on the blue screen and stored in memory at the location (PUNICODE_STRING) **KiBugCheckDriver**.

Cause

The driver is referencing session space addresses from the system process context. This probably results from the driver queuing an item to a system worker thread.

This driver needs to comply with Terminal Server's memory management rules.

Bug Check 0xD0: DRIVER_CORRUPTED_MMPOOL

7/2/2019 • 2 minutes to read • Edit Online

The DRIVER_CORRUPTED_MMPOOL bug check has a value of 0x000000D0. This indicates that the system attempted to access invalid memory at a process IRQL that was too high.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

DRIVER_CORRUPTED_MMPOOL Parameters

PARAMETER	DESCRIPTION
1	Memory referenced
2	IRQL at time of reference
3	**0:** Read **1:** Write
4	Address that referenced memory

Cause

The kernel attempted to access pageable memory (or perhaps completely invalid memory) when the IRQL was too high. The ultimate cause of this problem is almost certainly a driver that has corrupted the system pool.

In most cases, this bug check results if a driver corrupts a large allocation (PAGE_SIZE or larger). Smaller allocations result in **bug check 0xC5** (DRIVER_CORRUPTED_EXPOOL).

Resolution

If you have recently installed any new software, check to see if it is properly installed. Check for updated drivers on the manufacturer's website.

To debug this error, use the special pool option of Driver Verifier. If this fails to reveal the driver that caused the error, use the Global Flags utility to enable the special pool by pool tag.

For information about the special pool, consult the Driver Verifier section of the Windows Driver Kit.

An alternate method is to open the **\\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management** registry key. In this key, create or edit the **ProtectNonPagedPool** value, and set it equal to DWORD 1. Then reboot. Then the system will unmap all freed nonpaged pool. This will prevent drivers from corrupting the pool. (This does not protect the pool from DMA hardware, however.)

Bug Check 0xD1: DRIVER_IRQL_NOT_LESS_OR_EQUAL

10/15/2019 • 4 minutes to read • Edit Online

The DRIVER_IRQL_NOT_LESS_OR_EQUAL bug check has a value of 0x000000D1. This indicates that a kernel-mode driver attempted to access pageable memory while the process IRQL that was too high.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

DRIVER_IRQL_NOT_LESS_OR_EQUAL parameters

PARAMETER	DESCRIPTION
1	Memory referenced.
2	IRQL at time of reference.
3	• 0 - Read • 1 - Write • 2 - Execute • 8 - Execute
4	Address that referenced memory. Use **ln** (list nearest symbols) on this address to see the name of the function.

Cause

Typically, when this error occurs, a driver has tried to access an address that is pageable (or that is completely invalid) while the interrupt request level (IRQL) was too high. This can be caused by:

- Dereferencing a bad pointer (such as a NULL or freed pointer) while executing at or above DISPATCH_LEVEL.
- Accessing pageable data at or above DISPATCH_LEVEL.
- Executing pageable code at or above DISPATCH_LEVEL.

If a driver that is responsible for the error can be identified, its name is printed on the blue screen and stored in memory at the location (PUNICODE_STRING) **KiBugCheckDriver**. You can use **dx** (display debugger object model expression), a debugger command, to display this: **dx KiBugCheckDriver**.

This bug check is usually caused by drivers that have used improper memory addresses.

Possible causes for the page fault include the following events:

- The function was marked as pageable and was running at an elevated IRQL (which includes obtaining a lock).
- The function call was made to a function in another driver, and that driver was unloaded.
- The function was called by using a function pointer that was an invalid pointer.

Resolution

If the problem is caused by the driver that you are developing, make sure that the function that was executing at the time of the bug check is (1) not marked as pageable or (2) does not call any other inline functions that could be paged out.

The **!analyze** debugger extension displays information about the bug check and can be helpful in determining the root cause. The following example is output from **!analyze**.

```
DRIVER_IRQL_NOT_LESS_OR_EQUAL (d1)
An attempt was made to access a pageable (or completely invalid) address at an
interrupt request level (IRQL) that is too high.  This is usually
caused by drivers using improper addresses.
If kernel debugger is available get stack backtrace.
Arguments:
Arg1: fffff808add27150, memory referenced
Arg2: 0000000000000002, IRQL
Arg3: 0000000000000000, value 0 = read operation, 1 = write operation
Arg4: fffff808adc386a6, address which referenced memory
```

If a trap frame is available in the dump file, use the **.trap** command to set your context to the provided address.

To start debugging this type of bug check, examine the stack trace by using the **k**, **kb**, **kc**, **kd**, **kp**, **kP**, **kv** (display stack backtrace) commands.

In the debugger, run the **!irql** command to display information about the IRQL of a processor on the target computer before the debugger break. For example:

```
0: kd> !irql
Debugger saved IRQL for processor 0x0 -- 2 (DISPATCH_LEVEL)
```

In the majority of cases of this type of bug check, the issue is not the IRQL level, but rather the memory that is being accessed.

Because this bug check is usually caused by drivers that have used improper memory addresses, use parameters 1, 3, and 4 to investigate further.

Use **ln** (list nearest symbols) with parameter 4 to see the name of the function that was called. Also examine the **!analyze** output to see if faulting code is identified.

Use **!pool** on the parameter 1 address to see whether it is paged pool. Use **!address** and the advanced **!pte** command to learn more about this area of memory.

Use the display memory commands to examine the memory referenced in command in parameter 1.

Use the **u**, **ub**, **uu** (unassemble) commands to look at the code in the address which referenced the memory in parameter 4.

Use the command `lm t n` to list modules that are loaded in the memory. Use **!memusage** and to examine the general state of the system memory.

Driver Verifier

Driver Verifier is a tool that runs in real time to examine the behavior of drivers. For example, Driver Verifier checks the use of memory resources, such as memory pools. If it identifies errors in the execution of driver code, it proactively creates an exception to allow that part of the driver code to be further scrutinized. Driver Verifier Manager is built into Windows and is available on all Windows PCs.

To start Driver Verifier Manager, type **verifier** at a command prompt. You can configure which drivers to verify. The code that verifies drivers adds overhead as it runs, so try to verify the smallest number of drivers possible. For more information, see Driver Verifier.

Remarks

If you are not equipped to use the Windows debugger to work on this problem, you can use some basic troubleshooting techniques.

- Check the System Log in Event Viewer for additional error messages that might help identify the device or driver that is causing this bug check.
- If a driver is identified in the bug check message, disable the driver or check with the manufacturer for driver updates.
- Confirm that any new hardware that is installed is compatible with the installed version of Windows. For example, you can get information about required hardware at Windows 10 Specifications.

For additional general troubleshooting information, see Blue screen data.

Bug Check 0xD2: BUGCODE_ID_DRIVER

7/2/2019 • 2 minutes to read • Edit Online

The BUGCODE_ID_DRIVER bug check has a value of 0x000000D2. This indicates that a problem occurred with an NDIS driver.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

BUGCODE_ID_DRIVER Parameters

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	MESSAGE AND CAUSE
Address of the miniport block	Number of bytes requested	0	1	**Allocating shared memory at raised IRQL.** A driver called **NdisMAllocateSharedMemory** with IRQL >= DISPATCH_LEVEL.
Address of the miniport block	The *Status* value submitted to **NdisMResetComplete**	The *AddressingReset* value submitted to **NdisMResetComplete**	0	**Completing reset when one is not pending.** A driver called **NdisMResetComplete**, but no reset was pending.
Address of the miniport block	Memory page containing address being freed	Address of shared memory signature	Virtual address being freed	**Freeing shared memory not allocated.** A driver called **NdisMFreeSharedMemory** or **NdisMFreeSharedMemoryAsync** with an address that is not located in NDIS shared memory.
Address of the miniport block	Address of the packet that is incorrectly included in the packet array	Address of the packet array	Number of packets in the array	**Indicating packet not owned by it.** The miniport's packet array is corrupt.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	MESSAGE AND CAUSE
Address of the MiniBlock	Address of the driver object	0	0	**NdisAddDevice: AddDevice** called with a **MiniBlock** that is not on the **NdisMiniDriverList**.
Address of the MiniBlock	The MiniBlock's reference count	0	0	**NdisMUnload: MiniBlock** is getting unloaded but it is still on **NdisMiniDriverList**.
Address of the miniport block	Memory page	Wrapper context	Address of shared memory signature	**Overwrote past allocated shared memory.** The address being written to is not located in NDIS shared memory.

In the following instances of this bug check, the meaning of the parameters depends on the message and on the value of Parameter 4.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	MESSAGE AND CAUSE
Address of the miniport block	Address of the miniport interrupt	Address of the miniport timer queue	1	**Unloading without deregistering interrupt.** A miniport driver failed its initialization without deregistering its interrupt.
Address of the miniport block	Address of the miniport timer queue	Address of the miniport interrupt	2	**Unloading without deregistering interrupt.** A miniport driver did not deregister its interrupt during the halt process.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	MESSAGE AND CAUSE
Address of the miniport block	Address of the miniport interrupt	Address of the miniport timer queue	1	**Unloading without deregistering timer.** A miniport driver failed its initialization without successfully canceling all its timers.
Address of the miniport block	Address of the miniport timer queue	Address of the miniport interrupt	2	**Unloading without deregistering timer.** A miniport driver halted without successfully canceling all its timers.

Remarks

This bug check code only occurs on Windows 2000 and Windows XP. In Windows Server 2003 and later, the corresponding code is **bug check 0x7C** (BUGCODE_NDIS_DRIVER).

On the checked build of Windows, only the **Allocating Shared Memory at Raised IRQL** and **Completing Reset When One is Not Pending** instances of this bug check can occur. All the other instances of bug check 0xD2 are replaced with ASSERTs. See Breaking Into the Debugger for details.

Bug Check 0xD3: DRIVER_PORTION_MUST_BE_NONPAGED

11/22/2019 • 2 minutes to read • Edit Online

The DRIVER_PORTION_MUST_BE_NONPAGED bug check has a value of 0x000000D3. This indicates that the system attempted to access pageable memory at a process IRQL that was too high.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

DRIVER_PORTION_MUST_BE_NONPAGED Parameters

PARAMETER	DESCRIPTION
1	Memory referenced
2	IRQL at time of reference
3	**0:** Read **1:** Write
4	Address that referenced memory

If the driver responsible for the error can be identified, its name is printed on the blue screen and stored in memory at the location (PUNICODE_STRING) **KiBugCheckDriver**.

Cause

This bug check is usually caused by drivers that have incorrectly marked their own code or data as pageable.

Resolution

To begin debugging, use a kernel debugger to get a stack trace: the **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause, then use the **kb (Display Stack Backtrace)** command to get a stack trace.

Bug Check 0xD4: SYSTEM_SCAN_AT_RAISED_IRQL_CAUGHT_IMPROPER_DRIVER_UNLOAD

7/2/2019 • 2 minutes to read • Edit Online

The SYSTEM_SCAN_AT_RAISED_IRQL_CAUGHT_IMPROPER_DRIVER_UNLOAD bug check has a value of 0x000000D4. This indicates that a driver did not cancel pending operations before unloading.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

SYSTEM_SCAN_AT_RAISED_IRQL_CAUGHT_IMPROPER_DRIVER_UNLOAD Parameters

PARAMETER	DESCRIPTION
1	Memory referenced
2	IRQL at time of reference
3	**0:** Read **1:** Write
4	Address that referenced memory

If the driver responsible for the error can be identified, its name is printed on the blue screen and stored in memory at the location (PUNICODE_STRING) **KiBugCheckDriver**.

Cause

This driver failed to cancel lookaside lists, DPCs, worker threads, or other such items before unload. Subsequently, the system attempted to access the driver's former location at a raised IRQL.

Resolution

To begin debugging, use a kernel debugger to get a stack trace: the **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause, then use the **kb (Display Stack Backtrace)** command to get a stack trace. If the driver that caused the error has been identified, activate Driver Verifier and attempt to replicate this bug.

For full details on Driver Verifier, see the Windows Driver Kit.

Bug Check 0xD5: DRIVER_PAGE_FAULT_IN_FREED_SPECIAL_POOL

10/14/2019 • 2 minutes to read • Edit Online

The DRIVER_PAGE_FAULT_IN_FREED_SPECIAL_POOL bug check has a value of 0x000000D5. This indicates that a driver has referenced memory which was earlier freed.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

DRIVER_PAGE_FAULT_IN_FREED_SPECIAL_POOL Parameters

PARAMETER	DESCRIPTION
1	Memory address referenced
2	**0:** Read **1:** Write
3	Address that referenced memory (if known)
4	Reserved

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause. If the driver responsible for the error can be identified, its name is printed on the blue screen and stored in memory at the location (PUNICODE_STRING) **KiBugCheckDriver**.

Cause

The Driver Verifier **Special Pool** option has caught the driver accessing memory which was earlier freed.

For information about the special pool, consult the Driver Verifier section of the Windows Driver Kit.

Remarks

This cannot be protected by a **try - except** handler -- it can only be protected by a probe.

Bug Check 0xD6: DRIVER_PAGE_FAULT_BEYOND_END_OF_ALLOCATION

10/14/2019 • 2 minutes to read • Edit Online

The DRIVER_PAGE_FAULT_BEYOND_END_OF_ALLOCATION bug check has a value of 0x000000D6. This indicates the driver accessed memory beyond the end of its pool allocation.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

DRIVER_PAGE_FAULT_BEYOND_END_OF_ALLOCATION Parameters

PARAMETER	DESCRIPTION
1	Memory address referenced
2	**0:** Read **1:** Write
3	Address that referenced memory (if known)
4	Reserved

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause. If the driver responsible for the error can be identified, its name is printed on the blue screen and stored in memory at the location (PUNICODE_STRING) *KiBugCheckDriver*.

Cause

The driver allocated *n* bytes of memory and then referenced more than *n* bytes. The Driver Verifier **Special Pool** option detected this violation.

For information about the special pool, consult the Driver Verifier section of the Windows Driver Kit.

Remarks

This cannot be protected by a **try - except** handler -- it can only be protected by a probe.

Bug Check 0xD7: DRIVER_UNMAPPING_INVALID_VIEW

10/8/2019 • 2 minutes to read • Edit Online

The DRIVER_UNMAPPING_INVALID_VIEW bug check has a value of 0x000000D7. This indicates a driver is trying to unmap an address that was not mapped.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

DRIVER_UNMAPPING_INVALID_VIEW Parameters

PARAMETER	DESCRIPTION
1	Virtual address to unmap
2	**1:** The view is being unmapped **2:** The view is being committed
3	0
4	0

Remarks

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause. Use the **kb (Display Stack Backtrace)** command to get a stack trace: the driver that caused the error can be determined from the stack trace.

Bug Check 0xD8: DRIVER_USED_EXCESSIVE_PTES

7/2/2019 • 2 minutes to read • Edit Online

The DRIVER_USED_EXCESSIVE_PTES bug check has a value of 0x000000D8. This indicates that there are no more system page table entries (PTE) remaining.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

DRIVER_USED_EXCESSIVE_PTES Parameters

PARAMETER	DESCRIPTION
1	Pointer to the name of the driver that caused the error (Unicode string), or zero
2	Number of PTEs used by the driver that caused the error (if Parameter 1 is nonzero)
3	Total free system PTEs
4	Total system PTEs

If the driver responsible for the error can be identified, its name is printed on the blue screen and stored in memory at the location (PUNICODE_STRING) **KiBugCheckDriver**.

Cause

This is usually caused by a driver not cleaning up its memory use properly. Parameter 1 shows the driver which has consumed the most PTEs. The call stack will reveal which driver actually caused the bug check.

Resolution

Both drivers may need to be fixed. The total number of system PTEs may also need to be increased.

Bug Check 0xD9: LOCKED_PAGES_TRACKER_CORRUPTION

7/2/2019 • 2 minutes to read • Edit Online

The LOCKED_PAGES_TRACKER_CORRUPTION bug check has a value of 0x000000D9. This indicates that the internal locked-page tracking structures have been corrupted.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

LOCKED_PAGES_TRACKER_CORRUPTION Parameters

Parameter 1 indicates the type of violation. The meaning of the other parameters depends on the value of Parameter 1.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x01	The address of the internal lock tracking structure	The address of the memory descriptor list	The number of pages locked for the current process	The MDL is being inserted twice on the same process list.
0x02	The address of the internal lock tracking structure	The address of the memory descriptor list	The number of pages locked for the current process	The MDL is being inserted twice on the systemwide list.
0x03	The address of the first internal tracking structure found	The address of the internal lock tracking structure	The address of the memory descriptor list	The MDL was found twice in the process list when being freed.
0x04	The address of the internal lock tracking structure	The address of the memory descriptor list	0	The MDL was found in the systemwide list on free after it was removed.

Cause

The error is indicated by the value of Parameter 1.

Bug Check 0xDA: SYSTEM_PTE_MISUSE

10/23/2019 • 7 minutes to read • Edit Online

The SYSTEM_PTE_MISUSE bug check has a value of 0x000000DA. This indicates that a page table entry (PTE) routine has been used in an improper way.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

SYSTEM_PTE_MISUSE Parameters

Parameter 1 indicates the type of violation. The meaning of the other parameters depends on the value of Parameter 1.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x01	The address of the internal lock tracking structure	The address of the memory descriptor list	The address of the duplicate internal lock tracking structure	The mapping being freed is a duplicate.
0x02	The address of the internal lock tracking structure	The number of mappings that the system expects to free	The number of mappings that the driver is requesting to free	The number of mappings being freed is incorrect.
0x03	The address of the first internal tracking structure found	The mapping address that the system expects to free	The mapping address that the driver is requesting to free	The mapping address being freed is incorrect.
0x04	The address of the internal lock tracking structure	The page frame number that the system expects should be first in the MDL	The page frame number that is currently first in the MDL	The first page of the mapped MDL has changed since the MDL was mapped.
0x05	The address of the first internal tracking structure found	The virtual address that the system expects to free	The virtual address that the driver is requesting to free	The start virtual address in the MDL being freed has changed since the MDL was mapped.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x06	The MDL specified by the driver	The virtual address specified by the driver	The number of mappings to free (specified by the driver)	The MDL being freed was never (or is currently not) mapped.
0x07	The initial mapping	The number of mappings	Reserved	(Windows 2000 only) The mapping range is being double-allocated.
0x08	The initial mapping	The number of mappings the caller is freeing	The number of mappings the system thinks should be freed	(Windows 2000 only) The caller is asking to free an incorrect number of mappings.
0x09	The initial mapping	The number of mappings that the caller is freeing	The mapping index that the system thinks is already free	(Windows 2000 only) The caller is asking to free several mappings, but at least one of them is not allocated.
0x0A	**1:** The driver requested "bug check on failure" in the MDL. **0:** The driver did not request "bug check on failure" in the MDL.	The number of mappings that the caller is allocating	The type of mapping pool requested	(Windows 2000 only) The caller is asking to allocate zero mappings.
0x0B	The corrupt mapping	The number of mappings that the caller is allocating	The type of mapping pool requested	(Windows 2000 only) The mapping list was already corrupt at the time of this allocation. The corrupt mapping is located below the lowest possible mapping address.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x0C	The corrupt mapping	The number of mappings that the caller is allocating	The type of mapping pool requested	(Windows 2000 only) The mapping list was already corrupt at the time of this allocation. The corrupt mapping is located above the lowest possible mapping address.
0x0D	The initial mapping	The number of mappings that the caller is freeing	The type of mapping pool	(Windows 2000 only) The caller is trying to free zero mappings.
0x0E	The initial mapping	The number of mappings that the caller is freeing	The type of mapping pool	(Windows 2000 only) The caller is trying to free mappings, but the guard mapping has been overwritten.
0x0F	The non-existent mapping	The number of mappings that the caller is trying to free	The type of mapping pool being freed	(Windows 2000 only) The caller is trying to free a non-existent mapping. The non-existent mapping is located below the lowest possible mapping address.
0x10	The non-existent mapping	The number of mappings the caller is trying to free	The type of mapping pool being freed	(Windows 2000 only) The caller is trying to free a non-existent mapping. The non-existent mapping is located above the highest possible mapping address.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x11	The non-existent mapping	The number of mappings that the caller is trying to free	The type of mapping pool being freed	(Windows 2000 only) The caller is trying to free a non-existent mapping. The non-existent mapping is at the base of the mapping address space.
0x100	The number of mappings being requested	The caller's identifying tag	The address of the routine that called the caller of this routine	The caller requested 0 mappings.
0x101	The first mapping address	The caller's identifying tag	The owner's identifying tag	A caller is trying to free a mapping address range that it does not own.
0x102	The first mapping address	The caller's identifying tag	Reserved	The mapping address space that the caller is trying to free is apparently empty.
0x103	The address of the invalid mapping	The caller's identifying tag	The number of mappings in the mapping address space	The mapping address space that the caller is trying to free is still reserved. **MmUnmapReservedMapping** must be called before **MmFreeMappingAddress**.
0x104	The first mapping address	The caller's identifying tag	The owner's identifying tag	The caller is attempting to map an MDL to a mapping address space that it does not own.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x105	The first mapping address	The caller's identifying tag	Reserved	The caller is attempting to map an MDL to an invalid mapping address space. The caller has mostly likely specified an invalid address.
0x107	The first mapping address	The address of the non-empty mapping	The last mapping address	The caller is attempting to map an MDL to a mapping address space that has not been properly reserved. The caller should have called **MmUnmapReservedMapping** prior to calling **MmMapLockedPagesWithReservedMapping**
0x108	The first mapping address	The caller's identifying tag	The owner's identifying tag	The caller is attempting to unmap a locked mapping address space that it does not own.
0x109	The first mapping address	The caller's identifying tag	Reserved	The caller is attempting to unmap a locked virtual address space that is apparently empty.
0x10A	The first mapping address	The number of mappings in the locked mapping address space	The number of mappings to unmap	The caller is attempting to unmap more mappings than actually exist in the locked mapping address space.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x10B	The first mapping address	The caller's identifying tag	The number of mappings to unmap	The caller is attempting to unmap a portion of a locked virtual address space that is not currently mapped.
0x10C	The first mapping address	The caller's identifying tag	The number of mappings to unmap	The caller is not unmapping the entirety of the locked mapping address space.
0x200	The first mapping address	0	0	The caller is attempting to reserve a mapping address space that contains no mappings.
0x201 0x202	The first mapping address to reserve	The address of the mapping that has already been reserved	The number of mappings to reserve	One of the mappings that the caller is attempting to reserve has already been reserved.
0x300	The first mapping address to release	0	0	The caller is attempting to release a mapping address space that contains no mappings.
0x301	The address of the mapping	0	0	The caller is attempting to release a mapping that it is not permitted to release.
0x302	The address that the caller is trying to release.	Reserved	Reserved	The caller is attempting to release a system address that is not currently mapped.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x303	The first mapping address	The number of mappings to release	0	The caller is attempting to release a mapping address range that was not reserved.
0x304	The first mapping address	The number of mappings to release	0	The caller is attempting to release a mapping address range that begins in the middle of a different allocation.
0x305	The first mapping address	The number of mappings that the caller is trying to release	The number of mappings that should be released	The caller is attempting to release the wrong number of mappings.
0x306	The first mapping address	The free mapping address	The number of mappings to release	One of the mappings that the caller is attempting to release is already free.
0x400	The base address of the I/O space mapping	The number of pages to be freed	0	The caller is trying to free an I/O space mapping that the system is unaware of.

Cause

The error is indicated by the value of Parameter 1.

A stack trace will identify the driver that caused the error.

Bug Check 0xDB: DRIVER_CORRUPTED_SYSPTES

7/2/2019 • 2 minutes to read • Edit Online

The DRIVER_CORRUPTED_SYSPTES bug check has a value of 0x000000DB. This indicates that an attempt was made to touch memory at an invalid IRQL, probably due to corruption of system PTEs.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

DRIVER_CORRUPTED_SYSPTES Parameters

PARAMETER	DESCRIPTION
1	Memory referenced
2	IRQL
3	**0:** Read **1:** Write
4	Address in code which referenced memory

Cause

A driver tried to access pageable (or completely invalid) memory at too high of an IRQL. This bug check is almost always caused by drivers that have corrupted system PTEs.

Resolution

If this bug check occurs, the culprit can be detected by editing the registry. In the **\\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management** registry key, create or edit the **TrackPtes** value, and set it equal to DWORD 3. Then reboot. The system will then save stack traces, and if the driver commits the same error, the system will issue **bug check 0xDA** (SYSTEM_PTE_MISUSE). Then the stack trace will identify the driver that caused the error.

Bug Check 0xDC: DRIVER_INVALID_STACK_ACCESS

10/8/2019 • 2 minutes to read • Edit Online

The DRIVER_INVALID_STACK_ACCESS bug check has a value of 0x000000DC. This indicates that a driver accessed a stack address that lies below the stack pointer of the stack's thread.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

DRIVER_INVALID_STACK_ACCESS Parameters

None

Remarks

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause.

Bug Check 0xDE: POOL_CORRUPTION_IN_FILE_AREA

7/2/2019 • 2 minutes to read • Edit Online

The POOL_CORRUPTION_IN_FILE_AREA bug check has a value of 0x000000DE. This indicates that a driver has corrupted pool memory that is used for holding pages destined for disk.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

POOL_CORRUPTION_IN_FILE_AREA Parameters

None

Cause

When the Memory Manager dereferenced the file, it discovered this corruption in pool memory.

Bug Check 0xDF: IMPERSONATING_WORKER_THREAD

4/8/2019 • 2 minutes to read • Edit Online

The IMPERSONATING_WORKER_THREAD bug check has a value of 0x000000DF. This indicates that a workitem did not disable impersonation before it completed.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

IMPERSONATING_WORKER_THREAD Parameters

PARAMETER	DESCRIPTION
1	The worker routine that caused this error
2	The parameter passed to this worker routine
3	A pointer to the work item
4	Reserved

Cause

A worker thread was impersonating another process, and failed to disable impersonation before it returned.

Bug Check 0xE0: ACPI_BIOS_FATAL_ERROR

7/2/2019 • 2 minutes to read • Edit Online

The ACPI_BIOS_FATAL_ERROR bug check has a value of 0x000000E0. This indicates that one of your computer components is faulty.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

ACPI_BIOS_FATAL_ERROR Parameters

The parameters for this bug check are issued by the BIOS, not by Windows. They can only be interpreted by the hardware vendor.

Cause

Your computer's BIOS has reported that a component in the system is so faulty that there is no way for Windows to operate. The BIOS is indicating that there is no alternative but to issue a bug check.

Resolution

You can determine which component is faulty by running the diagnostic disk or tool that was included with your computer.

If you do not have this tool, you must contact the system vendor and report this error message to them. They will be able to help you correct this hardware problem. This enables Windows to operate.

Microsoft cannot address this error. Only the hardware vendor is qualified to analyze it.

Bug Check 0xE1: WORKER_THREAD_RETURNED_AT_BAD_IRQL

7/2/2019 • 2 minutes to read • Edit Online

The WORKER_THREAD_RETURNED_AT_BAD_IRQL bug check has a value of 0x000000E1. This indicates that a worker thread completed and returned with IRQL >= DISPATCH_LEVEL.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

WORKER_THREAD_RETURNED_AT_BAD_IRQL Parameters

PARAMETER	DESCRIPTION
1	Address of the worker routine
2	IRQL that the worker thread returned at
3	Work item parameter
4	Work item address

Cause

A worker thread completed and returned with IRQL >= DISPATCH_LEVEL.

Resolution

To find the driver that caused the error, use the **ln (List Nearest Symbols)** debugger command:

```
kd> ln address
```

where *address* is the worker routine address given in Parameter 1.

Bug Check 0xE2: MANUALLY_INITIATED_CRASH

7/2/2019 • 2 minutes to read • Edit Online

The MANUALLY_INITIATED_CRASH bug check has a value of 0x000000E2. This indicates that the user deliberately initiated a crash dump from either the kernel debugger or the keyboard.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

MANUALLY_INITIATED_CRASH Parameters

None

Remarks

For more information about manually-initiated crash dumps, see Forcing a System Crash. .

Bug Check 0xE3: RESOURCE_NOT_OWNED

7/2/2019 • 2 minutes to read • Edit Online

The RESOURCE_NOT_OWNED bug check has a value of 0x000000E3. This indicates that a thread tried to release a resource it did not own.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

RESOURCE_NOT_OWNED Parameters

PARAMETER	DESCRIPTION
1	Address of resource
2	Address of thread
3	Address of owner table (if it exists)
4	Reserved

Bug Check 0xE4: WORKER_INVALID

7/2/2019 • 2 minutes to read • Edit Online

The WORKER_INVALID bug check has a value of 0x000000E4. This indicates that memory that should not contain an executive work item does contain such an item, or that a currently active work item was queued.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

WORKER_INVALID Parameters

Parameter 1 indicates the code position.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x0	Address of work item	Start of pool block	End of pool block	An active worker item was freed.
0x1	Address of work item	Queue number	0	An active worker item was queued.
0x2	Address of work item	Address of I/O worker routine	0	A queued I/O worker item was freed.
0x3	Address of work item	Address of invalid object	0	An attempt was made to initialize an I/O worker item with an invalid object.
0x5	Address of work item	Queue number	NUMA Node targeted or -1 if all NODES were searched for.	An attempt was made to queue a work item before Worker Queued was initialized.
0x6	Address of work item	Queue number	0	Invalid queue type was provided.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x7	Address of work item	Queue number	0	An attempt was made to queue a work item with an invalid worker routine address.

Cause

This is usually caused by a driver freeing memory which still contains an executive work item.

Bug Check 0xE6: DRIVER_VERIFIER_DMA_VIOLATION

7/2/2019 • 7 minutes to read • Edit Online

The DRIVER_VERIFIER_DMA_VIOLATION bug check has a value of 0x000000E6. This is the bug check code for all Driver Verifier **DMA Verification** violations.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

DRIVER_VERIFIER_DMA_VIOLATION Parameters

Parameter 1 is the only parameter of interest. This parameter identifies the exact violation. If a debugger is attached, an informative message is displayed in the debugger.

PARAMETER 1	CAUSE OF ERROR AND DEBUGGER MESSAGE
0x00 - Miscellaneous DMA error.	This code can represent two kinds of errors as indicated by parameter 2: 0x1 - The driver tried to flush too many bytes to the end of the map register file. Parameter 3 - Number of bytes left in the MDL. Parameter 4 - Number of bytes left requested to be flushed. 0x2 - Windows has run out of contiguous map registers. Parameter 3 - Map registers needed. Parameter 4 - Number of contiguous map registers.
0x01	The performance counter has decreased. The old and new values of the counter are displayed.
0x02	The performance counter has increased too fast. The counter value is displayed in the debugger.
0x03	The driver freed too many DMA common buffers. Usually this means it freed the same buffer two times. Parameter 2 - Number of extra common buffers freed.
0x04	The driver freed too many DMA adapter channels. Usually this means it freed the same adapter channel two times. Parameter 2 - Number of extra adapter channels freed.

PARAMETER 1	CAUSE OF ERROR AND DEBUGGER MESSAGE
0x05	The driver freed too many DMA map registers. Usually this means it freed the same map register two times. Parameter 2 - Number of extra map registers freed.
0x06	The driver freed too many DMA scatter/gather lists. Usually this means it freed the same scatter/gather list two times. Parameter 2 - Allocated scatter-gather lists. Parameter 3 - Freed scatter-gather lists.
0x07	The driver tried to release the adapter without first freeing all its common buffers. Parameter 2 - Pointer to the DMA adapter. Parameter 3 - Number of outstanding common buffers. Parameter 4 - Pointer to the corresponding internal verifier data.
0x08	The driver tried to release the adapter without first freeing all adapter channels, common buffers, or scatter/gather lists. Parameter 2 - Pointer to the DMA adapter. Parameter 3 - Number of outstanding adapter channels. Parameter 4 - Pointer to the corresponding internal verifier data.
0x09	The driver tried to release the adapter without first freeing all map registers. Parameter 2 - Pointer to the DMA adapter. Parameter 3 - Number of outstanding map registers. Parameter 4 - Pointer to the corresponding internal verifier data.
0x0A	The driver tried to release the adapter without first freeing all its scatter/gather lists. Parameter 2 - Pointer to the DMA adapter. Parameter 3 - Number of outstanding scatter-gather lists. Parameter 4 - Pointer to the corresponding internal verifier data.
0x0B	The driver has allocated too many adapter channels at the same time (Only one adapter channel is permitted per adapter.) Parameter 2 - Outstanding adapter channels.

PARAMETER 1	CAUSE OF ERROR AND DEBUGGER MESSAGE
0x0C	The driver tried to allocate too many map registers at the same time. Parameter 2 - Required map registers. Parameter 3 - Maximum map registers.
0x0D	The driver did not flush its adapter buffers. Parameter 2 - Number of bytes mapped. Parameter 3 - Maximum number of bytes that can be mapped at a time.
0x0E	The driver tried a DMA transfer without locking the buffer. The buffer in question was in paged memory. Parameter 2 - Address of the DMA buffer MDL.
0x0F	The driver or the hardware wrote outside its allocated DMA buffer. Parameter 2 is the Violation code. 0x01 : The tag before the DMA buffer has been modified.Expected tag is DmaVrfy0. Parameter 3 - Buffer length. Parameter 4 - Buffer start. 0x02 : The tag after the DMA buffer has been modified. Expected tag is DmaVrfy0. Parameter 3 - Buffer length. Parameter 4 - Buffer start. 0x03 : Free map register was overwritten. Parameter 3 - Corruption address. Expected fill pattern is 0x0F. 0x04 : Padding before the buffer has been incorrectly modified. Parameter 3 - Buffer start. Expected padding is 0x0F. Parameter 4 - Corruption address. 0x05 : Padding after the buffer has been incorrectly modified. Parameter 3 - Buffer start. Parameter 4 - Corruption address. Expected padding pattern is 0x0F.
0x10	The driver tried to free its map registers while some were still mapped. Parameter 2 - Number of registers still mapped.

PARAMETER 1	CAUSE OF ERROR AND DEBUGGER MESSAGE
0x11	The driver has too many outstanding reference counts for the adapter. Parameter 2 - Reference count. Parameter 3 - Pointer to the DMA adapter. Parameter 4 - Pointer to the corresponding internal verifier data.
0x13	The driver called a DMA routine at an improper IRQL. Parameter 2 is the Violation code. 0x01 : Current IRQL is different than expected. Parameter 3 - Expected IRQL. Parameter 4 - Current IRQL. 0x02 : Current IRQL is higher than expected. Parameter 3 - Expected maximum IRQL. Parameter 4 - Current IRQL.
0x14	The driver called a DMA routine at an improper IRQL.
0x15	The driver tried to allocate too many map registers. Parameter 2 - Allocated map registers. Parameter 3 - Maximum map registers.
0x16	The driver tried to flush a buffer that is not mapped. Parameter 2 - Address in the system virtual space of the map register. Parameter 3 - Pointer to the corresponding internal verifier data.
0x18	The driver tried a DMA operation by using an adapter that was already released and no longer exists. Parameter 2 - Pointer to the DMA adapter. Parameter 3 - Pointer to the corresponding internal verifier data.
0x19	The driver passed a null DMA_ADAPTER value to a HAL routine.
0x1B	The driver passed an address and MDL to a HAL routine. However, this address is not within the bounds of this MDL. Parameter 2 - Virtual address that is out of MDL bounds. Parameter 3 - MDL.

PARAMETER 1	CAUSE OF ERROR AND DEBUGGER MESSAGE
0x1D	The driver tried to map an address range that was already mapped. Parameter 2 - Buffer to map start. Parameter 3 - Buffer to map end. Parameter 4 - System address in buffer that is already mapped.
0x1E	The driver called **HalGetAdapter**. This function is obsolete -- you must use **IoGetDmaAdapter** instead.
0x1F	Invalid DMA buffer. The driver referenced an invalid system address -- either before the first MDL, or after the end of the first MDL, or by using a transfer length that is longer than the MDL buffer and crosses a page boundary within the MDL.Parameter 2 is the Violation code. 0x01 : Virtual buffer address is before the first MDL. Parameter 3 - Virtual address of the start of the DMA buffer. Parameter 4 - Pointer to the first MDL describing the DMA buffer. 0x02 : Virtual address is after the first MDL. Parameter 3 - Virtual address of the start of the DMA buffer. Parameter 4 - Pointer to the first MDL describing the DMA buffer. 0x03 : Extra transfer length crosses a page boundary. Parameter 3 - Pointer to the MDL describing the DMA buffer. Parameter 4 - Length of the DMA transfer. 0x04 : Virtual address of a DMA buffer is not cache aligned. Parameter 3 - Virtual address of the start of the DMA buffer. Parameter 4 - Pointer to MDL describing the DMA buffer. 0x05 : DMA buffer length is not cache aligned. Parameter 3 - Length of the DMA buffer. Parameter 4 - Pointer to MDL describing the DMA buffer.
0x20	The driver tried to flush a map register that hasn't been mapped. Parameter 2 - Map register base. Parameter 3 - The VA of the start of the DMA buffer. Parameter 4 - Pointer to the MDL used to describe the DMA buffer.

PARAMETER 1	CAUSE OF ERROR AND DEBUGGER MESSAGE
0x21	The driver tried to map a zero-length buffer for transfer. Parameter 2 - Pointer to the corresponding internal verifier data.
0x22	DMA buffer not mapped in system VA. Parameter 2 - MDL
0x23	Cannot flush a channel that hasn't been completed or cancelled. Parameter 2 - Violation code. Value: 0x00 : Illegal channel flush Parameter 3 - Controller Id. Parameter 4 - Channel Number.
0x24	Insufficient buffer for requested length. Parameter 2 - Unaccounted length.
0x25	Unknown device description version.
0x26	IOMMU detected DMA violation. Parameter 2 - Device Object of faulting device. Parameter 3 - Faulting information (usually faulting physical address). Parameter 4 - Fault type (hardware specific).

Cause

See the description of each code in the Parameters section for a description of the cause.

Resolution

This bug check can only occur when Driver Verifier has been instructed to monitor one or more drivers. If you did not intend to use Driver Verifier, you should deactivate it. You might also consider removing the driver that caused this problem.

If you are the driver writer, use the information obtained through this bug check to fix the bugs in your code.

For more information on Driver Verifier, see Driver Verifier.

Bug Check 0xE7: INVALID_FLOATING_POINT_STATE

7/2/2019 • 2 minutes to read • Edit Online

The INVALID_FLOATING_POINT_STATE bug check has a value of 0x000000E7. This indicates that a thread's saved floating-point state is invalid.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

INVALID_FLOATING_POINT_STATE Parameters

Parameter 1 indicates which validity check failed. Parameter 4 is not used. The meaning of the other parameters depends on the value of Parameter 1.

PARAMETER 1	PARAMETER 2	PARAMETER 3	CAUSE OF ERROR
0x0	The flags field	0	The saved context flags field is invalid. Either FLOAT_SAVE_VALID is not set, or some reserved bits are nonzero.
0x1	The saved IRQL	The current IRQL	The current processor's IRQL is not the same as when the floating-point context was saved.
0x2	The saved address of the thread that owns this floating-point context	The current thread	The saved context does not belong to the current thread.

Cause

While restoring the previously-saved floating-point state for a thread, the state was found to be invalid.

Parameter 1 indicates which validity check failed.

Bug Check 0xE8: INVALID_CANCEL_OF_FILE_OPEN

7/2/2019 • 2 minutes to read • Edit Online

The INVALID_CANCEL_OF_FILE_OPEN bug check has a value of 0x000000E8. This indicates that an invalid file object was passed to **IoCancelFileOpen**.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

INVALID_CANCEL_OF_FILE_OPEN Parameters

PARAMETER	DESCRIPTION
1	The file object passed to **IoCancelFileOpen**
2	The device object passed to **IoCancelFileOpen**
3	Reserved
4	Reserved

Cause

The file object passed to **IoCancelFileOpen** is invalid. It should have reference of one. The driver that called **IoCancelFileOpen** is at fault.

Bug Check 0xE9: ACTIVE_EX_WORKER_THREAD_TERMINATION

7/2/2019 • 2 minutes to read • Edit Online

The ACTIVE_EX_WORKER_THREAD_TERMINATION bug check has a value of 0x000000E9. This indicates that an active executive worker thread is being terminated.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

ACTIVE_EX_WORKER_THREAD_TERMINATION Parameters

PARAMETER	DESCRIPTION
1	The exiting ETHREAD
2	Reserved
3	Reserved
4	Reserved

Cause

An executive worker thread is being terminated without having gone through the worker thread rundown code. This is forbidden; work items queued to the **ExWorkerQueue** must not terminate their threads.

A stack trace should indicate the cause.

Bug Check 0xEA: THREAD_STUCK_IN_DEVICE_DRIVER

7/2/2019 • 2 minutes to read • Edit Online

The THREAD_STUCK_IN_DEVICE_DRIVER bug check has a value of 0x000000EA. This indicates that a thread in a device driver is endlessly spinning.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

THREAD_STUCK_IN_DEVICE_DRIVER Parameters

PARAMETER	DESCRIPTION
1	A pointer to the stuck thread object
2	A pointer to the DEFERRED_WATCHDOG object
3	A pointer to the offending driver name
4	**In the kernel debugger:** The number of times the "intercepted" bug check 0xEA was hit **On the blue screen:** 1

Cause

A device driver is spinning in an infinite loop, most likely waiting for hardware to become idle.

This usually indicates problem with the hardware itself, or with the device driver programming the hardware incorrectly. Frequently, this is the result of a bad video card or a bad display driver.

Resolution

Use the **.thread (Set Register Context)** command together with Parameter 1. Then use **kb (Display Stack Backtrace)** to find the location where the thread is stuck.

If the kernel debugger is already connected and running when Windows detects a time-out condition. Then **DbgBreakPoint** will be called instead of **KeBugCheckEx**. A detailed message will be printed to the debugger. See Sending Output to the Debuggefor more information.

This message will include what would have been the bug check parameters. Because no actual bug check was issued, the **.bugcheck (Display Bug Check Data)** command will not be useful. The four parameters can also be retrieved from Watchdog's global variables by using **dd watchdog!g_WdBugCheckData L5**" on a 32-bit system, or **dq watchdog!g_WdBugCheckData L5**" on a 64-bit system.

Debugging this error in an interactive manner such as this will enable you to find an offending thread, set breakpoints in it, and then use **g (Go)** to return to the spinning code to debug it further.

On multiprocessor machines (OS build 3790 or earlier), you can hit a time out if the spinning thread is interrupted by a hardware interrupt and an ISR or DPC routine is running at the time of the bug check. This is because the time out's work item can be delivered and handled on the second CPU and the same time. If this occurs, you must look deeper at the offending thread's stack to determine the spinning code which caused the time out to occur. Use the **dds (Display Words and Symbols)** command to do this.

Bug Check 0xEB: DIRTY_MAPPED_PAGES_CONGESTION

7/2/2019 • 2 minutes to read • Edit Online

The DIRTY_MAPPED_PAGES_CONGESTION bug check has a value of 0x000000EB. This indicates that no free pages are available to continue operations.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

DIRTY_MAPPED_PAGES_CONGESTION Parameters

PARAMETER	DESCRIPTION
1	The total number of dirty pages
2	The number of dirty pages destined for the page file
3	Windows Server 2003 only: The size of the nonpaged pool available at the time of the bug check (in pages) Windows Vista and later versions: Reserved
4	Windows Server 2003 only: The number of transition pages that are currently stranded Windows Vista and later versions: The most recent modified write error status

Cause

The file system driver stack has deadlocked and most of the modified pages are destined for the file system. Because the file system is non-operational, the system has crashed because none of the modified pages can be reused without losing data. Any file system or filter driver in the stack may be at fault.

To see general memory statistics, use the **!vm 3** extension.

This bug check can occur for any of the following reasons:

- A driver has blocked, deadlocking the modified or mapped page writers. Examples of this include mutex deadlocks or accesses to paged out memory in file system drivers or filter drivers. This indicates a driver bug.

  If Parameter 1 or Parameter 2 is large, this is a possibility. Use **!vm 3**.

- A storage driver is not processing requests. Examples of this are stranded queues and unresponsive drives. This indicates a driver bug.

If Parameter 1 or Parameter 2 is large, this is a possibility. Use **!process 0 7**.

- Windows Server 2003 only: Not enough pool is available for the storage stack to write out modified pages. This indicates a driver bug.

  If Parameter 3 is small, this is a possibility. Use **!vm** and **!poolused 2**.

Bug Check 0xEC: SESSION_HAS_VALID_SPECIAL_POOL_ON_EXIT

7/2/2019 • 2 minutes to read • Edit Online

The SESSION_HAS_VALID_SPECIAL_POOL_ON_EXIT bug check has a value of 0x000000EC. This indicates that a session unload occurred while a session driver still held memory.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

SESSION_HAS_VALID_SPECIAL_POOL_ON_EXIT Parameters

PARAMETER	DESCRIPTION
1	The session ID
2	The number of special pool pages that are leaking
3	Reserved
4	Reserved

Cause

This error is caused by a session driver not freeing its special pool allocations prior to a session unload. This indicates a bug in win32k.sys, atmfd.dll, rdpdd.dll, or a video driver.

Bug Check 0xED: UNMOUNTABLE_BOOT_VOLUME

7/2/2019 • 2 minutes to read • Edit Online

The UNMOUNTABLE_BOOT_VOLUME bug check has a value of 0x000000ED. This indicates that the I/O subsystem attempted to mount the boot volume and it failed.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

UNMOUNTABLE_BOOT_VOLUME Parameters

PARAMETER	DESCRIPTION
1	The device object of the boot volume
2	The status code from the file system that describes why it failed to mount the volume
3	Reserved
4	Reserved

Resolution

If you are debugging this error, use the !analyze -v extension. This extension displays relevant data specific error to the error.

This bug check is typically related to the failure of the OS Boot storage device such as hard drive. To attempt to validate the file system and the recover the boot record the following troubleshooting steps may be helpful.

1. In Windows 10, use Troubleshoot > Advanced Options > Startup Repair. You may need to create bootable recovery media and boot from a USB drive or DVD to run the Windows Recovery Environment.
2. From the command prompt in the Windows Recovery Environment use CHKDSK /r to attempt to repair the file system.
3. Use the bootrec command to fix master and boot records.

If these steps are not successful it is possible that the hard drive has failed. Some hard drive vendors provide diagnostic tools that may help confirm a hardware failure.

(Developer Content) Bug Check 0xEF: CRITICAL_PROCESS_DIED

7/2/2019 • 4 minutes to read • Edit Online

The CRITICAL_PROCESS_DIED bug check has a value of 0x000000EF. This indicates that a critical system process died. A critical process is one that forces the system to bug check if it terminates. This can happen when the state of the process is corrupted or otherwise is damaged. When this happens, as these processes are critical to the operation of Windows, a system bug check occurs as the operating system integrity is in question.

Built in Windows critical system services include csrss.exe, wininit.exe, logonui.exe, smss.exe, services.exe, conhost.exe, and winlogon.exe.

A developer can also create a service and set its recovery option to Restart the Computer for more information see Set up Recovery Actions to Take Place When a Service Fails.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

CRITICAL_PROCESS_DIED Parameters

PARAMETER	DESCRIPTION
1	The process object
2	If this is 0, a process died. If this is 1, a thread died.
3	Reserved
4	Reserved

Resolution

Determining the cause of this issues typically requires the use of the debugger to gather additional information. Multiple dump files should be examined to see if this stop code has similar characteristics, such as the code that is running when the stop code appears.

For more information, see Crash dump analysis using the Windows debuggers (WinDbg), Using the !analyze Extension and !analyze.

In many cases a user dump is also created before the system bugchecks. In general, when a user dump is available, it should be examined first to root cause the issue. This is because there are limitations to debugging user mode code from the kernel dump, including paged out/missing data. For more information see, User-Mode Dump Files.

Consider using the event log to see if there are errors that occur leading up to this stop code. If there are, these errors can be used to examine specific services or other code to investigate.

If the bug check can be reproduced at will, consider using Time Travel Tracing to record events leading up to the bug check. For more information see, Time Travel Debugging - Overview.

Once information about the code in question is available, set a breakpoint in the related code before this code is executed and single step forward through the code looking at the values of critical variables that are used to control the code flow. Carefully examine this area of your code to look for false assumptions or other mistakes.

Use the second parameter of the bug check to determine if a dying process or thread caused the bug check.

If it is a process, use the !process command to display information on the process before and after the point of failure to look for abnormal behavior. The Process explorer utility can used to gather general information about which process are running and parent child relationships.

If it is a thread, consider using the !thread command to display information about the thread. For information about threads in kernel mode, see Changing Contexts.

For general information on threads and process as well as additional specifics on Windows protected, critical code such as wininit and csrss, see *Windows Internals* by Pavel Yosifovich, Mark E. Russinovich, David A. Solomon, and Alex Ionescu.

General Troubleshooting Tips

If you are not able to work with the debugger, these general troubleshooting tips may be helpful.

- If you recently added hardware to the system, try removing or replacing it. Or check with the manufacturer to see if any patches are available.
- If new device drivers or system services have been added recently, try removing or updating them. Try to determine what changed in the system that caused the new bug check code to appear.
- Check the System Log in Event Viewer for additional error messages that might help pinpoint the device or driver that is causing the error. For more information, see Open Event Viewer. Look for critical errors in the system log that occurred in the same time window as the blue screen.
- Check with the manufacturer to see if an updated system BIOS or firmware is available.
- You can try running the hardware diagnostics supplied by the system manufacturer.
- Confirm that any new hardware that is installed is compatible with the installed version of Windows. For example, you can get information about required hardware at Windows 10 Specifications.
- Run a virus detection program. Viruses can infect all types of hard disks formatted for Windows, and resulting disk corruption can generate system bug check codes. Make sure the virus detection program checks the Master Boot Record for infections.
- Use the System File Checker tool to repair missing or corrupted system files. The System File Checker is a utility in Windows that allows users to scan for corruptions in Windows system files and restore corrupted files. Use the following command to run the System File Checker tool (SFC.exe).

  ```
  SFC /scannow
  ```

  For more information, see Use the System File Checker tool to repair missing or corrupted system files.
- Look in **Device Manager** to see if any devices are marked with the exclamation point (!). Review the events log displayed in driver properties for any faulting driver. Try updating the related driver.

See also

Crash dump analysis using the Windows debuggers (WinDbg)

Bug Check 0xF0: STORAGE_MINIPORT_ERROR

7/2/2019 • 2 minutes to read • Edit Online

The STORAGE_MINIPORT_ERROR bug check has a value of 0x00000F0. It indicates that a storage Miniport driver failed to complete a SRB request.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

STORAGE_MINIPORT_ERROR Parameters

PARAMETER	DESCRIPTION
1	Error Code. See Values below.
2	See Values below.
3	See Values below.
4	See Values below.

Values

```
1: Miniport failed to complete a SRB request even after successful reset operation.
    2 - Driver name unicode string address
    3 - SRB address
    4 - Storport unit device object

2: Miniport failed to complete a SRB request even after successful abort operation for the SRB.
    2 - Driver name unicode string address
    3 - Abort SRB address
    4 - SRB that was being aborted

3: Miniport failed to complete a request within a given timeout.
    2 - Driver name unicode string address
    3 - SRB address
    4 - Timeout of the request

4: Miniport failed to complete a request for a crypto operation. This can occur if it is trying to enable an
encryption key on an ICE (Inline Crypto Engine) enabled UFS host.
    2 - Driver name unicode string address
    3 - The STOR_CRYPTO_OPERATION_TYPE for this failure, typically StorCryptoOperationInsertKey
    4 - Reserved
```

Cause

A bug in the storage Miniport driver kept a SRB request from completing. See the error code values listed above for the specific type of failure.

Resolution

The !analyze debug extension displays information about the bug check and can be helpful in determining the root cause.

The driver name returned in parameter 2 should point to the offending driver.

See Also

Bug Check Code Reference

Storport's Interface with Storport Miniport Drivers

Bug Check 0xF1: SCSI_VERIFIER_DETECTED_VIOLATION

10/23/2019 • 2 minutes to read • Edit Online

The SCSI_VERIFIER_DETECTED_VIOLATION bug check has a value of 0x000000F1. This is the bug check code for all Driver Verifier **SCSI Verification** violations.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

SCSI_VERIFIER_DETECTED_VIOLATION Parameters

Parameter 1 identifies the type of violation.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x1000	First argument passed	Second argument passed	Reserved	The miniport driver passed bad arguments to **ScsiPortInitialize**.
0x1001	Delay, in microseconds	Reserved	Reserved	The miniport driver called **ScsiPortStallExecution** and specified a delay greater than 0.1 second, stalling the processor too long.
0x1002	Address of routine that took too long	Address of miniport's HW_DEVICE_EXTENSION	Duration of the routine, in microseconds	A miniport routine called by the port driver took longer than 0.5 second to execute. (0.5 seconds is the limit for most routines. However, the **HwInitialize** routine is allowed 5 seconds, and the **FindAdapter** routine is exempt.)

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x1003	Address of miniport's HW_DEVICE_EXTENSION	Address of the SRB	Reserved	The miniport driver completed a request more than once.
0x1004	Address of the SRB	Address of miniport's HW_DEVICE_EXTENSION	Reserved	The miniport driver completed a request with an invalid SRB status.
0x1005	Address of miniport's HW_DEVICE_EXTENSION	Address of LOGICAL_UNIT_EXTENSION	Reserved	The miniport driver called **ScsiPortNotification** to ask for **NextLuRequest**, but an untagged request is still active.
0x1006	Address of miniport's HW_DEVICE_EXTENSION	Invalid virtual address	Reserved	The miniport driver passed an invalid virtual address to **ScsiPortGetPhysicalAddress**. (This usually means the address supplied doesn't map to the common buffer area.)
0x1007	Address of ADAPTER_EXTENSION	Address of miniport's HW_DEVICE_EXTENSION	Reserved	The reset hold period for the bus ended, but the miniport driver still has outstanding requests.
0x2001	Delay, in microseconds	Reserved	Reserved	The Storport miniport driver called **StorPortStallExecution** and specified a delay longer than 0.1 second, stalling the processor for an excessive length of time.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x2002	Reserved	Reserved	Reserved	**StorPortGetUncachedExtension** was not called from the miniport driver's **HwStorFindAdapter** routine. The **StorPortGetUncachedExtension** routine can only be called from the miniport driver's **HwStorFindAdapter** routine and only for a bus-master adapter. A Storport miniport driver must set the **SrbExtensionSize** of the **HW_INITIALIZATION_DATA** (Storport) structure before calling **StorPortGetUncachedExtension**.
0x2003	Reserved	Reserved	Reserved	An invalid address was passed to the **StorPortGetDeviceBase** routine. The **StorPortGetDeviceBase** routine supports only those addresses that were assigned to the driver by the system Plug and Play (PnP) manager.
0x2004	Reserved	Reserved	Reserved	The Storport miniport driver completed the same I/O request more than once.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x2005	Reserved	Reserved	Reserved	The Storport miniport driver passed an invalid virtual address to one of the **StorPortRead***xxx* or **StorPortWrite***xxx* routines. This usually means the address supplied doesn't map to the common buffer area. The specified *Register* or *Port* must be in mapped memory-space range returned by **StorPortGetDeviceBase** routine.

Cause

See the description of each code in the Parameters section for an explanation of the cause.

Resolution

This bug check can only occur when Driver Verifier has been instructed to monitor one or more drivers. If you did not intend to use Driver Verifier, you should deactivate it. You might consider removing the driver which caused this problem as well.

If you are the driver writer, use the information obtained through this bug check to fix the bugs in your code.

The Driver Verifier **SCSI Verification** option is available only in Windows XP and later. The Driver Verifier **Storport Verification** option is available only in Windows 7 and later. For full details on Driver Verifier, see the Windows Driver Kit.

Bug Check 0xF2: HARDWARE_INTERRUPT_STORM

7/2/2019 • 2 minutes to read • Edit Online

The HARDWARE_INTERRUPT_STORM bug check has a value of 0x000000F2. This indicates that the kernel detected an interrupt storm.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

HARDWARE_INTERRUPT_STORM Parameters

PARAMETER	DESCRIPTION
1	Address of the ISR (or first ISR in the chain) connected to the storming interrupt vector
2	ISR context value
3	Address of the interrupt object for the storming interrupt vector
4	0x1 if the ISR is not chained, 0x2 if the ISR is chained

Cause

This bugcheck indicates that the kernel has detected an interrupt storm. An interrupt storm is defined as a level triggered interrupt signal staying in the asserted state. This is fatal to the system in the manner that the system will hard hang, or "bus lock".

This can happen because of the following:

- A piece of hardware does not release its interrupt signal after being told to do so by the device driver.
- A device driver does not instruct its hardware to release the interrupt signal because it does not believe the interrupt was initiated from its hardware.
- A device driver claims the interrupt even though the interrupt was not initiated from its hardware. Note that this can only occur when multiple devices are sharing the same IRQ.
- The ELCR (edge level control register) is set incorrectly.
- Edge and Level interrupt triggered devices share an IRQ.

All of these cases will instantly hard hang your system. Instead of hard hanging the system, this bugcheck is initiated since in many cases it can identify the culprit.

When the bugcheck occurs, the module containing the ISR (interrupt service routine) of the storming IRQ is displayed on the screen. This is an example of what you would see:

```
*** STOP: 0x000000F2 (0xFCA7C55C, 0x817B9B28, 0x817D2AA0, 0x00000002)
An interrupt storm has caused the system to hang.
*** Address FCA7C55C base at FCA72000, Datestamp 3A72BDEF - ACPI.sys
```

In the event the fourth parameter is a 0x00000001, the module pointed to is very likely the culprit. Either the driver is broken, or the hardware is malfunctioning.

In the event the fourth parameter is a 0x00000002, the module pointed to is the first ISR in the chain, and is never guaranteed to be the culprit.

Resolution

A user experiencing this bugcheck repeatedly should try to isolate the problem by looking for devices that are on the same IRQ as the one for which the module is a driver for (in this case, the same IRQ that ACPI is using).

Bug Check 0xF3: DISORDERLY_SHUTDOWN

7/2/2019 • 2 minutes to read • Edit Online

The DISORDERLY_SHUTDOWN bug check has a value of 0x000000F3. This indicates that Windows was unable to shut down due to lack of memory.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

DISORDERLY_SHUTDOWN Parameters

PARAMETER	DESCRIPTION
1	The total number of dirty pages
2	The number of dirty pages destined for the page file
3	Windows Server 2003 only: The size of the nonpaged pool available at the time of the bug check (in pages) Windows Vista and later: Reserved
4	Windows Server 2003 only: The current shut down stage Windows Vista and later: The most recent modified write error status

Cause

Windows attempted to shut down, but there were no free pages available to continue operations.

Because applications were not terminated and drivers were not unloaded, they continued to access pages even after the modified writer had terminated. This causes the system to run out of pages, since the page files could be used.

Bug Check 0xF4: CRITICAL_OBJECT_TERMINATION

7/2/2019 • 2 minutes to read • Edit Online

The CRITICAL_OBJECT_TERMINATION bug check has a value of 0x000000F4. This indicates that a process or thread crucial to system operation has unexpectedly exited or been terminated.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

CRITICAL_OBJECT_TERMINATION Parameters

PARAMETER	DESCRIPTION
1	The terminating object type: **0x3:** Process **0x6:** Thread
2	The terminating object
3	The process image file name
4	Pointer to an ASCII string containing an explanatory message

Cause

Several processes and threads are necessary for the operation of the system. When they are terminated for any reason, the system can no longer function.

Resolution

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause.

Bug Check 0xF5: FLTMGR_FILE_SYSTEM

7/2/2019 • 2 minutes to read • Edit Online

The FLTMGR_FILE_SYSTEM bug check has a value of 0x000000F5. This indicates that an unrecoverable failure occurred in the Filter Manager.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

FLTMGR_FILE_SYSTEM Parameters

Parameter 1 indicates the type of violation. The meaning of the other parameters depends on the value of Parameter 1.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x66	Pointer to the callback data structure for the operation.	0	0	The minifilter returned FLT_PREOP_SUCCESS_WITH_CALLBACK or FLT_PREOP_SYNCHRONIZE from a preoperation callback, but did not register a corresponding postoperation callback.
0x67	Pointer to the callback data structure for the operation.	0	Error NTSTATUS code for the operation	An internal object ran out of space, and the system is unable to allocate new space.
0x68	Reserved	Address of the FLT_FILE_NAME_INFORMATIONN structure	Reserved	A FLT_FILE_NAME_INFORMATION structure was dereferenced too many times.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x6A	File object pointer for the file.	0	0	The file-open or file-create request could not be canceled, because one or more handles have been created for the file.
0x6B	Frame ID	0	Thread	Invalid BACKPOCKET IRPCTRL state.
0x6C	Frame ID	BackPocket List	Thread	Too many nested PageFaults for BACKPOCKETED IRPCTR.
0x6D	Address of the minifilter's context structure	Address of the CONTEXT_NODE structure	0	The context structure was dereferenced too many times. This means that the reference count on the Filter Manager's CONTEXT_NODE structure went to zero while it was still attached to its associated object.
0x6E	Address of the minifilter's context structure	Address of the CONTEXT_NODE structure	0	The context structure was referenced after being freed.

Cause

The cause of the problem is indicated by the value of Parameter 1. See the table in the Parameters section.

Resolution

If Parameter 1 equals **0x66**, you can debug this problem by verifying that your minifilter driver has registered a post-operation callback for this operation. The current operation can be found in the callback data structure. (See Parameter 2.) Use the **!fltkd.cbd** debugger extension.

If Parameter 1 equals **0x67**, you should verify that you do not have a nonpaged pool leak somewhere in the system.

If Parameter 1 equals **0x6A**, make sure that your minifilter driver does not reference this file object (see Parameter 2) to get a handle at any point during your minifilter's processing of this operation.

If Parameter 1 equals **0x6B** or **0x6C**, then a non-recoverable internal state error has occurred which will cause the operating system to bug check.

If Parameter 1 equals **0x6D**, make sure that your minifilter driver does not call **FltReleaseContext** too many times for the given context (see Parameter 2).

If Parameter 1 equals 0x6E, make sure that your minifilter driver does not call **FltReferenceContext** after the given context has been deleted (see Parameter 2).

Bug Check 0xF6: PCI_VERIFIER_DETECTED_VIOLATION

7/2/2019 • 2 minutes to read • Edit Online

The PCI_VERIFIER_DETECTED_VIOLATION bug check has a value of 0x000000F6. This indicates that an error occurred in the BIOS or another device being verified by the PCI driver.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

PCI_VERIFIER_DETECTED_VIOLATION Parameters

Parameter 1 is the only parameter of interest; this identifies the nature of the failure detected.

PARAMETER 1	CAUSE OF ERROR
0x01	An active bridge was reprogrammed by the BIOS during a docking event.
0x02	The PMCSR register was not updated within the spec-mandated time.
0x03	A driver has written to Windows-controlled portions of a PCI device's configuration space.

Cause

The PCI driver detected an error in a device or BIOS being verified.

Bug Check 0xF7: DRIVER_OVERRAN_STACK_BUFFER

7/2/2019 • 2 minutes to read • Edit Online

The DRIVER_OVERRAN_STACK_BUFFER bug check has a value of 0x000000F7. This indicates that a driver has overrun a stack-based buffer.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

DRIVER_OVERRAN_STACK_BUFFER Parameters

PARAMETER	DESCRIPTION
1	The actual security check cookie from the stack
2	The expected security check cookie
3	The bit-complement of the expected security check cookie
4	0

Cause

A driver overran a stack-based buffer (or local variable) in a way that would have overwritten the function's return address and jumped back to an arbitrary address when the function returned.

This is the classic "buffer overrun" hacking attack. The system has been brought down to prevent a malicious user from gaining complete control of it.

Resolution

Use the **kb (Display Stack Backtrace)** command to get a stack trace.

The last routine on the stack before the buffer overrun handlers and bug check call is the one that overran its local variable.

Bug Check 0xF8: RAMDISK_BOOT_INITIALIZATION_FAILED

7/2/2019 • 2 minutes to read • Edit Online

The RAMDISK_BOOT_INITIALIZATION_FAILED bug check has a value of 0x000000F8. This indicates that an initialization failure occurred while attempting to boot from the RAM disk.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

RAMDISK_BOOT_INITIALIZATION_FAILED Parameters

PARAMETER	DESCRIPTION
1	Indicates the cause of the failure. **1:** No LoaderXIPRom descriptor was found in the loader memory list. **2:** Unable to open the RAM disk driver (ramdisk.sys or \Device\Ramdisk). **3:** FSCTL_CREATE_RAM_DISK failed. **4:** Unable to create GUID string from binary GUID. **5:** Unable to create symbolic link pointing to the RAM disk device.
2	NTSTATUS code
3	0
4	0

Bug Check 0xF9: DRIVER_RETURNED_STATUS_REPARSE_FOR_VOLUME_OPEN

10/8/2019 • 2 minutes to read • Edit Online

The DRIVER_RETURNED_STATUS_REPARSE_FOR_VOLUME_OPEN bug check has a value of 0x000000F9. This indicates that a driver returned STATUS_REPARSE to an IRP_MJ_CREATE request with no trailing names.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

DRIVER_RETURNED_STATUS_REPARSE_FOR_VOLUME_OPEN Parameters

PARAMETER	DESCRIPTION
1	The device object that was opened
2	The device object to which the IRP_MJ_CREATE request was issued
3	Address of the Unicode string containing the new name of the file (to be reparsed)
4	Information returned by the driver for the IRP_MJ_CREATE request

Remarks

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause. STATUS_REPARSE should be returned only for IRP_MJ_CREATE requests with trailing names, as that indicates the driver is supporting name spaces.

Bug Check 0xFA: HTTP_DRIVER_CORRUPTED

7/2/2019 • 2 minutes to read • Edit Online

The HTTP_DRIVER_CORRUPTED bug check has a value of 0x000000FA. This indicates that the HTTP kernel driver (Http.sys) has reached a corrupted state and cannot recover.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

HTTP_DRIVER_CORRUPTED Parameters

Parameter 1 identifies the exact state of the HTTP kernel driver.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x1	Address of work item	Name of the file that contains the work item check	Line number of the work item check within the file	A work item is invalid. This will eventually result in thread pool corruption and an access violation.

Bug Check 0xFC: ATTEMPTED_EXECUTE_OF_NOEXECUTE_MEMORY

7/2/2019 • 2 minutes to read • Edit Online

The ATTEMPTED_EXECUTE_OF_NOEXECUTE_MEMORY bug check has a value of 0x000000FC. This indicates that an attempt was made to execute non-executable memory.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

ATTEMPTED_EXECUTE_OF_NOEXECUTE_MEMORY Parameters

PARAMETER	DESCRIPTION
1	The virtual address whose execution was attempted
2	The contents of the page table entry (PTE)
3	Reserved
4	Reserved

Resolution

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause. When possible, the Unicode string of the driver name that attempted to execute non-executable memory is printed on the bug check screen and is also saved in **KiBugCheckDriver**. Otherwise, the driver in question can often be found by running a stack trace and then reviewing the current instruction pointer.

Bug Check 0xFD: DIRTY_NOWRITE_PAGES_CONGESTION

7/2/2019 • 2 minutes to read • Edit Online

The DIRTY_NOWRITE_PAGES_CONGESTION bug check has a value of 0x000000FD. This indicates that there are no free pages available to continue basic system operations.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

DIRTY_NOWRITE_PAGES_CONGESTION Parameters

PARAMETER	DESCRIPTION
1	Total number of dirty pages
2	Number of non-writeable dirty pages
3	Reserved
4	Most recently modified write-error status

Cause

This bug check usually occurs because the component that owns the modified non-writeable pages failed to write out these pages after marking the relevant files as "do not write" to memory management. This indicates a driver bug.

Resolution

For more information about which driver is causing the problem, use the **!vm 3** extension, followed by **!memusage 1** .

Bug Check 0xFE: BUGCODE_USB_DRIVER

7/2/2019 • 2 minutes to read • Edit Online

The BUGCODE_USB_DRIVER bug check has a value of 0x000000FE. This indicates that an error has occurred in a universal serial bus (USB) driver.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

BUGCODE_USB_DRIVER Parameters

The four bug check parameters are displayed on the bug check stop screen and available using !analyze. Parameter 1 identifies the type of violation.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x1	Reserved	Reserved	Reserved	An internal error has occurred in the USB stack.
0x2	Address of the pending IRP	Address of the IRP that was passed in	Address of the USB request block (URB) that caused the error	The USB client driver has submitted a URB that is still attached to another IRP pending in the bus driver.
0x3	Reserved	Reserved	Reserved	The USB miniport driver has generated a bug check. This usually happens in response to a hardware failure.
0x4	Address of the IRP	Address of the URB	Reserved	The caller has submitted an IRP that is already pending in the USB bus driver.
0x5	Device extension pointer of the host controller	PCI vendor, product id for the controller	Pointer to endpoint data structure	A hardware failure has occurred because of a bad physical address found in a hardware data structure.
0x6	Object address	Signature that was expected	Reserved	An internal data structure (object) is corrupted.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x7	Pointer to usbport.sys debug log	Message string	File name	See the provided message string for detailed information.
0x8	1	Reserved	Reserved	Reserved
	2	Device object	IRP	An IRP was received by the hub driver that it does not expect or has not registered for.
	3	Reserved	Reserved	Reserved
	4	PDO if Parameter 3 is not NULL. Context if Parameter 3 is NULL.	Context or NULL	Fatal PDO trap
	5	Reserved	Reserved	Reserved
	6	Time-out code. See the table below.	Time-out code context: port data	Fatal time-out

If Parameter 1 has a value of 8 and Parameter 2 has a value of 6, then Parameter 3 is a time-out code. Possible values for the time-out code are given in the following table.

TIME-OUT CODE	MEANING
0	Non-fatal time-out
1	Failed resuming a suspended port.
2	Timed out waiting for a reset, initiated by a client driver, to complete before suspending the port.
3	Timed out waiting for the port to complete resume before suspending it.
4	Timed out waiting for the port-change state machine to be disabled prior to suspending the port.
5	Timed out waiting for a suspend-port request to complete.
6	Timed out waiting for the port-change state machine to be disabled.

TIME-OUT CODE	MEANING
7	Timed out waiting for the port-change state machine to be closed.
8	Timed out waiting for the hub to resume from selective suspend.
9	Timed out waiting for the hub to resume from selective suspend prior to system suspend.
10	Timed out waiting for port-change state machine to become idle.

Resolution

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause.

Bug Check 0xFF: RESERVE_QUEUE_OVERFLOW

7/2/2019 • 2 minutes to read • Edit Online

The RESERVE_QUEUE_OVERFLOW bug check has a value of 0x000000FF. This indicates that an attempt was made to insert a new item into a reserve queue, causing the queue to overflow.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

RESERVE_QUEUE_OVERFLOW Parameters

PARAMETER	DESCRIPTION
1	The address of the reserve queue
2	The size of the reserve queue
3	0
4	0

Resolution

The **!analyze** debug extension displays information about the bug check and can be very helpful in determining the root cause.

Bug Check 0x100: LOADER_BLOCK_MISMATCH

7/2/2019 • 2 minutes to read • Edit Online

The LOADER_BLOCK_MISMATCH bug check has a value of 0x00000100. This indicates that either the loader block is invalid, or it does not match the system that is being loaded.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

LOADER_BLOCK_MISMATCH Parameters

PARAMETER	DESCRIPTION
1	3
2	The size of the loader black extension
3	The major version of the loader block
4	The minor version of the loader block

Bug Check 0x101: CLOCK_WATCHDOG_TIMEOUT

7/2/2019 • 2 minutes to read • Edit Online

The CLOCK_WATCHDOG_TIMEOUT bug check has a value of 0x00000101. This indicates that an expected clock interrupt on a secondary processor, in a multi-processor system, was not received within the allocated interval.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

CLOCK_WATCHDOG_TIMEOUT Parameters

PARAMETER	DESCRIPTION
1	Clock interrupt time-out interval, in nominal clock ticks
2	0
3	The address of the processor control block (PRCB) for the unresponsive processor
4	0

Cause

The specified processor is not processing interrupts. Typically, this occurs when the processor is nonresponsive or is deadlocked.

Bug Check 0x102: DPC_WATCHDOG_TIMEOUT

7/2/2019 • 2 minutes to read • Edit Online

The DPC_WATCHDOG_TIMEOUT bug check has a value of 0x00000102. This indicates that The DPC watchdog routine was not executed within the allocated time interval.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

DPC_WATCHDOG_TIMEOUT Parameters

PARAMETER	DESCRIPTION
1	DPC watchdog time out interval in nominal clock ticks.
2	The PRCB address of the hung processor.
3	Reserved
4	Reserved

Cause

This bug check typically means that either an ISR is hung at an IRQL that is below clock level and above dispatch level, or a DPC routine is hung on the specified processor.

For example for StorPort Miniport drivers, StorPort.sys handles I/O completions in a routine that runs at DISPATCH_LEVEL and that serially calls the I/O completion routines of all IRPs that have just completed. If I/O completion routines singly or together take too much time, the keyboard and/or mouse may stop responding. It is also possible that the Windows DPC Watchdog timer routine will decide that the StorPort routine has taken excessive time to finish.

Resolution

A kernel driver in the storage stack can reduce the problem's likelihood by efficient coding of the driver's I/O completion routine. If it is still not possible to do all necessary processing in the completion routine in enough time, the routine can create a work element for the I/O work, queue up the element to a work queue and return STATUS_MORE_PROCESSING_REQUIRED; a worker thread of the driver should then find the work element, do the work and do IoCallerDriver for the IRP to ensure the IRP's further I/O processing.

Bug Check 0x103: MUP_FILE_SYSTEM

7/2/2019 • 2 minutes to read • Edit Online

The MUP_FILE_SYSTEM bug check has a value of 0x00000103. This bug check indicates that the multiple UNC provider (MUP) has encountered invalid or unexpected data. As a result, the MUP cannot channel a remote file system request to a network redirector, the Universal Naming Convention (UNC) provider.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

MUP_FILE_SYSTEM Parameters

Parameter 1 identifies the type of violation.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x1	The address of the pending IRP.	The address of the file object whose file context could not be found.	The address of the device object.	The MUP could not locate the file context that corresponds to a file object. This typically indicates that the MUP is seeing an I/O request for a file object for which MUP did not see a corresponding IRP_MJ_CREATE request. The likely cause of this bug check is a filter driver error.
0x2	The address of the expected file context.	The address that was actually retrieved from the file object.	Reserved	A file context is known to exist for the file object, but was not what was expected (for example, it might be **NULL**).

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x3	The address of the IRP context.	The IRP completion status code.	The driver object of the UNC provider that completed the IRP (might be **NULL**).	The IRP completion status was unexpected or invalid. This bug check occurs only when you are using a Checked Build of Windows and should only be caused by file system filter drivers that are attached to legacy network redirectors. Legacy redirectors use **FsRtlRegisterUncProvider** to register with MUP. This bug check detects filter drivers that return an NTSTATUS that is not STATUS_SUCCESS in IRP_MJ_CLEANUP or IRP_MJ_CLOSE requests.
0x4	Address of the IRP	Address of the file object	The file context for the file object	An I/O operation was started on a file object before the create request for the file object was completed.

Remarks

The MUP maintains context information on a per-file object basis for all file objects it handles.

Bug Check 0x104: AGP_INVALID_ACCESS

7/2/2019 • 2 minutes to read • Edit Online

The AGP_INVALID_ACCESS bug check has a value of 0x00000104. This indicates that the GPU wrote to a range of Accelerated Graphics Port (AGP) memory that had not previously been committed.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

AGP_INVALID_ACCESS Parameters

PARAMETER	DESCRIPTION
1	Offset (in ULONG) within the AGP verifier page to the first ULONG data that is corrupted
2	0
3	0
4	0

Cause

Typically, this bug check is caused by an unsigned or improperly tested video driver. It can also be caused by an old BIOS.

Resolution

Check for display driver and computer BIOS updates.

Bug Check 0x105: AGP_GART_CORRUPTION

7/2/2019 • 2 minutes to read • Edit Online

The AGP_GART_CORRUPTION bug check has a value of 0x00000105. This indicates that the Graphics Aperture Remapping Table (GART) is corrupt.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

AGP_GART_CORRUPTION Parameters

PARAMETER	DESCRIPTION
1	The base address (virtual) of the GART
2	The offset into the GART where the corruption occurred
3	The base address (virtual) of the GART cache (a copy of the GART)
4	0

Cause

This bug check is typically caused by improper direct memory access (DMA) by a driver.

Resolution

Enable Driver Verifier for any unsigned drivers. Remove them or disable them one by one until the erring driver is identified.

Bug Check 0x106: AGP_ILLEGALLY_REPROGRAMMED

7/2/2019 • 2 minutes to read • Edit Online

The AGP_ILLEGALLY_REPROGRAMMED bug check has a value of 0x00000106. This indicates that the Accelerated Graphics Port (AGP) hardware has been reprogrammed by an unauthorized agent.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

AGP_ILLEGALLY_REPROGRAMMED Parameters

PARAMETER	DESCRIPTION
1	The originally programmed AGP command register value
2	The current command register value
3	0
4	0

Cause

This bug check is typically caused by an unsigned, or improperly tested, video driver.

Resolution

Check the video manufacturer's Web site for updated display drivers or use VGA mode.

Bug Check 0x108: THIRD_PARTY_FILE_SYSTEM_FAILURE

7/2/2019 • 2 minutes to read • Edit Online

The THIRD_PARTY_FILE_SYSTEM_FAILURE bug check has a value of 0x00000108. This indicates that an unrecoverable problem has occurred in a third-party file system or file system filter.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

THIRD_PARTY_FILE_SYSTEM_FAILURE Parameters

PARAMETER	DESCRIPTION
1	Identifies the file system that failed. Possible values include: **1:** Polyserve (Psfs.sys)
2	The address of the exception record.
3	The address of the context record.
4	Reserved.

Cause

One possible cause of this bug check is disk corruption. Corruption in the third-party file system or bad blocks (sectors) on the hard disk can induce this error. Corrupted SCSI and IDE drivers can also adversely affect the Windows operating system's ability to read and write to disk, thus causing the error.

Another possible cause is depletion of nonpaged pool memory. If the nonpaged pool is completely depleted, this error can stop the system.

Resolution

To debug this problem: Use the **.cxr (Display Context Record)** command with Parameter 3, and then use **kb (Display Stack Backtrace)**.

To resolve a disk corruption problem: Check Event Viewer for error messages from SCSI, IDE, or other disk controllers in the system that might help pinpoint the device or driver that is causing the error. Try disabling any virus scanners, backup programs, or disk defragmenter tools that continually monitor the system. You should also run hardware diagnostics supplied by the file system or the file system filter manufacturer.

To resolve a nonpaged pool memory depletion problem: Add new physical memory to the computer. This

will increase the quantity of nonpaged pool memory available to the kernel.

Bug Check 0x109: CRITICAL_STRUCTURE_CORRUPTION

7/2/2019 • 3 minutes to read • Edit Online

The CRITICAL_STRUCTURE_CORRUPTION bug check has a value of 0x00000109. This indicates that the kernel has detected critical kernel code or data corruption.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

CRITICAL_STRUCTURE_CORRUPTION Parameters

PARAMETER	DESCRIPTION
1	Reserved
2	Reserved
3	Reserved
4	The type of the corrupted region. (See the following table later on this page.)

The value of Parameter 4 indicates the type of corrupted region.

PARAMETER 4	TYPE OF CORRUPTED REGION, TYPE OF CORRUPTION, OR TYPE OF ACTION TAKEN THAT CAUSED THE CORRUPTION
0x0	A generic data region
0x1	A function modification or the Itanium-based function location
0x2	A processor interrupt dispatch table (IDT)
0x3	A processor global descriptor table (GDT)
0x4	A type-1 process list corruption
0x5	A type-2 process list corruption

PARAMETER 4	TYPE OF CORRUPTED REGION, TYPE OF CORRUPTION, OR TYPE OF ACTION TAKEN THAT CAUSED THE CORRUPTION
0x6	A debug routine modification
0x7	A critical MSR modification
0x8	Object type
0x9	A processor IVT
0xA	Modification of a system service function
0xB	A generic session data region
0xC	Modification of a session function or .pdata
0xD	Modification of an import table
0xE	Modification of a session import table
0xF	Ps Win32 callout modification
0x10	Debug switch routine modification
0x11	IRP allocator modification
0x12	Driver call dispatcher modification
0x13	IRP completion dispatcher modification
0x14	IRP deallocator modification
0x15	A processor control register
0x16	Critical floating point control register modification
0x17	Local APIC modification

PARAMETER 4	TYPE OF CORRUPTED REGION, TYPE OF CORRUPTION, OR TYPE OF ACTION TAKEN THAT CAUSED THE CORRUPTION
0x18	Kernel notification callout modification
0x19	Loaded module list modification
0x1A	Type 3 process list corruption
0x1B	Type 4 process list corruption
0x1C	Driver object corruption
0x1D	Executive callback object modification
0x1E	Modification of module padding
0x1F	Modification of a protected process
0x20	A generic data region
0x21	A page hash mismatch
0x22	A session page hash mismatch
0x23	Load config directory modification
0x24	Inverted function table modification
0x25	Session configuration modification
0x26	An extended processor control register
0x27	Type 1 pool corruption
0x28	Type 2 pool corruption
0x29	Type 3 pool corruption
0x101	General pool corruption

PARAMETER 4	TYPE OF CORRUPTED REGION, TYPE OF CORRUPTION, OR TYPE OF ACTION TAKEN THAT CAUSED THE CORRUPTION
0x102	Modification of win32k.sys

Cause

There are generally three different causes for this bug check:

1. A driver has inadvertently, or deliberately, modified critical kernel code or data. Microsoft Windows Server 2003 with Service Pack 1 (SP1) and later versions of Windows for x64-based computers do not allow the kernel to be patched except through authorized Microsoft-originated hot patches. For more information, see Patching Policy for x64-based Systems.
2. A developer attempted to set a normal kernel breakpoint using a kernel debugger that was not attached when the system was started. Normal breakpoints (**bp**) can only be set if the debugger is attached at start time. Processor breakpoints (**ba**) can be set at any time.
3. A hardware corruption occurred. For example, the kernel code or data could have been stored in memory that failed.

Resolution

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause.

To start, examine the stack trace using the **k, kb, kc, kd, kp, kP, kv (Display Stack Backtrace)** command. You can specify the processor number to examine the stacks on all processors.

You can also set a breakpoint in the code leading up to this stop code and attempt to single step forward into the faulting code.

For more information see the following topics:

Crash dump analysis using the Windows debuggers (WinDbg)

If you are not equipped to use the Windows debugger to work on this problem, you can use some basic troubleshooting techniques.

- Check the System Log in Event Viewer for additional error messages that might help identify the device or driver that is causing this bug check.
- If a driver is identified in the bug check message, disable the driver or check with the manufacturer for driver updates.
- Run the Windows Memory Diagnostics tool, to test the memory. In the control panel search box, type Memory, and then click **Diagnose your computer's memory problems**. After the test is run, use Event viewer to view the results under the System log. Look for the *MemoryDiagnostics-Results* entry to view the results.
- You can try running the hardware diagnostics supplied by the system manufacturer.
- Confirm that any new hardware that is installed is compatible with the installed version of Windows. For example, you can get information about required hardware at Windows 10 Specifications.
- For additional general troubleshooting information, see **Blue Screen Data**.

Bug Check 0x10A: APP_TAGGING_INITIALIZATION_FAILED

7/2/2019 • 2 minutes to read • Edit Online

The APP_TAGGING_INITIALIZATION_FAILED bug check has a value of 0x0000010A.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Bug Check 0x10C: FSRTL_EXTRA_CREATE_PARAMETER_VIOLATION

7/2/2019 • 2 minutes to read • Edit Online

The FSRTL_EXTRA_CREATE_PARAMETER_VIOLATION bug check has a value of 0x0000010C. This indicates that a violation was detected in the File system Run-time library (FsRtl) Extra Create Parameter (ECP) package.

IMPORTANT

This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

FSRTL_EXTRA_CREATE_PARAMETER_VIOLATION Parameters

PARAMETER	DESCRIPTION
1	The type of violation. (See the following table later on this page for more details).
2	0
3	The address of the ECP.
4	The starting address of the ECP list.

The value of Parameter 1 indicates the type of violation.

PARAMETER 1	TYPE OF VIOLATION
0x1	The ECP signature is invalid, due to either a bad pointer or memory corruption.
0x2	The ECP has undefined flags set.
0x3	The ECP was not allocated by the FsRtl.
0x4	The ECP has flags set that are illegal for a parameter passed by a create caller.
0x5	The ECP is corrupted; its size is smaller than the header size.

PARAMETER 1	TYPE OF VIOLATION
0x6	The ECP that is being freed has non-empty list pointers; it might still be part of an ECP list.
0x11	The ECP list signature is invalid, due to either a bad pointer or memory corruption.
0x12	The ECP list has undefined flags set.
0x13	The ECP list was not allocated by the FsRtl.
0x14	The ECP list has flags set that are illegal for a parameter list passed by a create caller.
0x15	The ECP list passed by the create caller is empty.

Bug Check 0x10D: WDF_VIOLATION

7/2/2019 • 4 minutes to read • Edit Online

The WDF_VIOLATION bug check has a value of 0x0000010D. This indicates that Kernel-Mode Driver Framework (KMDF) detected that Windows found an error in a framework-based driver.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

WDF_VIOLATION Parameters

Parameter 1 indicates the specific error code of the bug check. Parameter 4 is reserved.

PARAMETER 1	PARAMETER 2	PARAMETER 3	CAUSE OF ERROR
0x1	Pointer to a WDF_POWER_ROUTINE_TIMED_OUT_DATA structure	Reserved	A framework-based driver has timed out during a power operation. This typically means that the device stack did not set the DO_POWER_PAGABLE bit and a driver attempted a pageable operation after the paging device stack was powered down.
0x2	Reserved	Reserved	An attempt is being made to acquire a lock that is currently being held.
0x3	WDFREQUEST handle	The number of outstanding references that remain on both buffers	Windows Driver Framework Verifier has encountered a fatal error. In particular, an I/O request was completed, but a framework request object cannot be deleted because there are outstanding references to the input buffer, the output buffer, or both.

PARAMETER 1	PARAMETER 2	PARAMETER 3	CAUSE OF ERROR
0x4	Reserved	The caller's address	A **NULL** parameter was passed to a function that required a non-**NULL** value.
0x5	The handle value passed in	Reserved	A framework object handle of the incorrect type was passed to a framework object method.
0x6			See table below.
0x7	The handle of the framework object	Reserved	A driver attempted to delete a framework object incorrectly by calling **WdfObjectDereference** to delete a handle instead of calling **WdfObjectDelete**.
0x8	The handle of the DMA transaction object	Reserved	An operation occurred on a DMA transaction object while it was not in the correct state.
0x9			Currently unused.
0xA	A pointer to a WDF_QUEUE_FATAL_ERROR_DATA structure	Reserved	A fatal error has occurred while processing a request that is currently in the queue.
0xB			See table below.
0xC	WDFDEVICE handle	Pointer to new PnP IRP	A new state-changing PnP IRP arrived while the driver was processing another state-changing PnP IRP.

PARAMETER 1	PARAMETER 2	PARAMETER 3	CAUSE OF ERROR
0xD	WDFDEVICE handle	Pointer to power IRP	A device's power policy owner received a power IRP that it did not request. There might be multiple power policy owners, but only one is allowed. A KMDF driver can change power policy ownership by calling **WdfDeviceInitSetPowerPolicyOwnership**.
0xE	IRQL at which the event callback function was called.	IRQL at which the event callback function returned.	An event callback function did not return at the same IRQL at which it was called. The callback function changed the IRQL directly or indirectly (for example, by acquiring a spinlock, which raises IRQL to DISPATCH_LEVEL, but not releasing the spinlock).
0xF	Address of an event callback function.	Reserved	An event callback function entered a critical region, but it did not leave the critical region before returning.

Parameter 1 is equal to 0x6

If Parameter 1 is equal to 0x6, then a fatal error was made in handling a WDF request. In this case, Parameter 2 further specifies the type of fatal error that has been made, as defined by the enumeration WDF_REQUEST_FATAL_ERROR.

PARAMETER 2	PARAMETER 3	CAUSE OF ERROR
0x1	The address of the IRP	No more I/O stack locations are available to format the underlying IRP.
0x2	The WDF request handle value	An attempt was made to format a framework request object that did not contain an IRP.
0x3	The WDF request handle value	The driver attempted to send a framework request that has already been sent to an I/O target.

PARAMETER 2	PARAMETER 3	CAUSE OF ERROR
0x4	A pointer to a WDR_REQUEST_FATAL_ERROR_INFORMATION_LENGTH_MISMATCH_DATA structure that contains a pointer to the IRP, a WDF request handle value, an IRP major function, and the number of bytes attempted to be written	The driver has completed a framework request, but has written more bytes to the output buffer than are specified in the IRP.

Parameter 1 is equal to 0xB

If Parameter 1 is equal to 0xB, then an attempt to acquire or release a lock was invalid. In this case, Parameter 3 further specifies the error that has been made.

PARAMETER 2	PARAMETER 3	CAUSE OF ERROR
The handle value	0x0	A handle passed to **WdfObjectAcquireLock** or **WdfObjectReleaseLock** represents an object that does not support synchronization locks.
A WDF spin lock handle	0x1	The spin lock is being released by a thread that did not acquire it.

Cause

See the description of each code in the Parameters section for an explanation of the cause.

Resolution

The **!analyze** debug extension displays information about the bug check and can be helpful in gathering information, such as the faulting code module.

Typically, the WDF dump file will yield further information on the driver that caused this bug check. Use this command to look at the log file.

```
kd> !wdfkd.wdflogdump <WDF_Driver_Name>
```

If Parameter 1 is equal to **0x2**, examine the caller's stack to determine the lock in question.

If Parameter 1 is equal to **0x3**, the driver's Kernel-Mode Driver Framework error log will include details about the outstanding references.

If Parameter 1 is equal to **0x4**, use the **ln debugger** command with the value of *Parameter 3* as its argument to determine which function requires a non-**NULL** parameter.

If Parameter 1 is equal to **0x7**, use the **!wdfkd.wdfhandle***Parameter 2* extension command to determine the handle type.

If Parameter 1 is equal to **0xA**, then the WDF_QUEUE_FATAL_ERROR_DATA structure will indicate either the problematic request or the queue handle. It will also indicate the NTSTATUS, if not STATUS_SUCCESS, when

available.

Bug Check 0x10E: VIDEO_MEMORY_MANAGEMENT_INTERNAL

7/2/2019 • 2 minutes to read • Edit Online

The VIDEO_MEMORY_MANAGEMENT_INTERNAL bug check has a value of 0x0000010E. This indicates that the video memory manager has encountered a condition that it is unable to recover from.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

VIDEO_MEMORY_MANAGEMENT_INTERNAL Parameters

Parameter 1 is the only parameter of interest; this identifies the exact violation. Values for Parameter 1 that do not appear in this table must be individually examined.

PARAMETER 1	CAUSE OF ERROR
0x1	An attempt was made to rotate a non-rotate range.
0x2	An attempt was made to destroy a non-empty process heap.
0x3	An attempt to unmap from an aperture segment failed.
0x4	A rotation in a must-succeed path failed.
0x5	A deferred command failed.
0x6	An attempt was made to reallocate resources for an allocation that was having its eviction canceled.
0x7	An invalid attempt was made to defer free usage.
0x8	The split direct memory access (DMA) buffer contains an invalid reference.
0x9	An attempt to evict an allocation failed.
0xA	An invalid attempt to use a pinned allocation was made.

PARAMETER 1	CAUSE OF ERROR
0xB	A driver returned an invalid error code from *BuildPagingBuffer*.
0xC	A resource leak was detected in a segment.
0xD	A segment is being used improperly.
0xE	An attempt to map an allocation into an aperture segment failed.
0xF	A driver returned an invalid error code from *AcquireSwizzlingRange*.
0x10	A driver returned an invalid error code from *ReleaseSwizzlingRange*.
0x11	An invalid attempt to use an aperture segment was made.
0x12	A driver overflowed the provided DMA buffer.
0x13	A driver overflowed the provided private data buffer.
0x14	An attempt to purge all segments failed.
0x15	An attempt was made to free a virtual address descriptor (VAD) that was still in the rotated state
0x16	A driver broke the guaranteed DMA buffer model contract.
0x17	An unexpected system command failure occurred.
0x18	An attempt to release a pinned allocation's resource failed.
0x19	A driver failed to patch a DMA buffer.
0x1A	The owner of a shared allocation was freed.
0x1B	An attempt was made to release an aperture range that is still in use.

PARAMETER 1	CAUSE OF ERROR
0x25	The GPU attempted to write over an undefined area of the aperture.

Cause

This bug check is usually caused by a video driver behaving improperly.

Resolution

If the problem persists, check Windows Update for an updated video driver.

Bug Check 0x10F: RESOURCE_MANAGER_EXCEPTION_NOT_HANDLED

7/2/2019 • 2 minutes to read • Edit Online

The RESOURCE_MANAGER_EXCEPTION_NOT_HANDLED bug check has a value of 0x0000010F. This indicates that the kernel transaction manager detected that a kernel-mode resource manager has raised an exception in response to a direct call-back. The resource manager is in an unexpected and unrecoverable state.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

RESOURCE_MANAGER_EXCEPTION_NOT_HANDLED Parameters

PARAMETER	DESCRIPTION
1	The address of the exception record
2	The address of the context record
3	The address of the exception code
4	The address of the resource manager

Bug Check 0x111: RECURSIVE_NMI

7/2/2019 • 2 minutes to read • Edit Online

The RECURSIVE_NMI bug check has a value of 0x00000111. This bug check indicates that a non-maskable-interrupt (NMI) occurred while a previous NMI was in progress.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Remarks

This bug check occurs when there is an error in the system management interrupt (SMI) code, and an SMI interrupts an NMI and enables interrupts. Execution then continues with NMIs enabled, and another NMI interrupts the NMI in progress.

Bug Check 0x112: MSRPC_STATE_VIOLATION

7/2/2019 • 2 minutes to read • Edit Online

The MSRPC_STATE_VIOLATION bug check has a value of 0x00000112. This indicates that the Msrpc.sys driver has initiated a bug check.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

MSRPC_STATE_VIOLATION Parameters

Parameters 1 and 2 are the only parameters of interest. Parameter 1 indicates the state violation type; the value for Parameter 2 is determined by the value of Parameter 1.

PARAMETER 1	PARAMETER 2	CAUSE OF ERROR
0x01	The exception code	A non-continuable exception was continued by the caller.
0x02	The error	The advanced local procedure call (ALPC) returned an invalid error.
0x03	The session to the server	The caller unloaded the Microsoft remote procedure call (MSRPC) driver while it was still in use. It is likely that open binding handles remain.
0x04 and 0x05	The session to the server	An invalid close command was received from the ALPC.
0x06	The binding handle	An attempt was made to bind a remote procedure call (RPC) handle a second time.
0x07	The binding handle	An attempt was made to perform an operation on a binding handle that was not bound.
0x08	The binding handle	An attempt was made to set security information on a binding handle that was already bound.

PARAMETER 1	PARAMETER 2	CAUSE OF ERROR
0x09	The binding handle	An attempt was made to set an option on a binding handle that was already bound.
0x0A	The call object	An attempt was made to cancel an invalid asynchronous remote procedure call.
0x0B	The call object	An attempt was made to push on an asynchronous pipe call when it was not expected.
0x0C and 0x0E	The pipe object	An attempt was made to push on an asynchronous pipe without waiting for notification.
0x0F	The pipe object	An attempt was made to synchronously terminate a pipe a second time.
0x15	The object closest to the error	An RPC internal error occurred.
0x16	Reserved	Two causally ordered calls were issued in an order that cannot be enforced by the RPC.
0x17	The call object	A server manager routine did not unsubscribe from notifications prior to completing the call.
0x18	The async handle	An invalid operation on the asynchronous handle occurred.

Cause

The most common cause of this bug check is that the caller of the Msrpc.sys driver violated the state semantics for such a call.

Bug Check 0x113: VIDEO_DXGKRNL_FATAL_ERROR

7/2/2019 • 2 minutes to read • Edit Online

The VIDEO_DXGKRNL_FATAL_ERROR bug check has a value of 0x00000113. This indicates that the Microsoft DirectX graphics kernel subsystem has detected a violation.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Bug Check 0x114: VIDEO_SHADOW_DRIVER_FATAL_ERROR

7/2/2019 • 2 minutes to read • Edit Online

The VIDEO_SHADOW_DRIVER_FATAL_ERROR bug check has a value of 0x00000114. This indicates that the shadow driver has detected a violation.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Bug Check 0x115: AGP_INTERNAL

7/2/2019 • 2 minutes to read • Edit Online

The AGP_INTERNAL bug check has a value of 0x00000115. This indicates that the accelerated graphics port (AGP) driver has detected a violation.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Bug Check 0x116: VIDEO_TDR_FAILURE

10/23/2019 • 6 minutes to read • Edit Online

The VIDEO_TDR_FAILURE bug check has a value of 0x00000116. This indicates that an attempt to reset the display driver and recover from a timeout failed.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

VIDEO_TDR_FAILURE Parameters

PARAMETER	DESCRIPTION
1	The pointer to the internal TDR recovery context, if available.
2	A pointer into the responsible device driver module (for example, the owner tag).
3	The error code of the last failed operation, if available.
4	Internal context dependent data, if available.

Cause

A common stability problem in graphics occurs when the system appears completely frozen or hung while processing an end-user command or operation. Usually the GPU is busy processing intensive graphics operations, typically during game-play. No screen updates occur, and users assume that their system is frozen. Users usually wait a few seconds and then reboot the system by pressing the power button. Windows tries to detect these problematic hang situations and dynamically recover a responsive desktop.

This process of detection and recovery is known as Timeout Detection and Recovery (TDR). The default timeout is 2 seconds. In the TDR process for video cards, the operating system's GPU scheduler calls the display miniport driver's *DxgkDdiResetFromTimeout* function to reinitialize the driver and reset the GPU.

During this process, the operating system tells the driver not to access the hardware or memory and gives it a short time for currently running threads to complete. If the threads do not complete within the timeout, then the system bug checks with 0x116 VIDEO_TDR_FAILURE. For more information, see Thread Synchronization and TDR.

The system can also bug check with VIDEO_TDR_FAILURE if a number of TDR events occur in a short period of time, by default more than five TDRs in one minute.

If the recovery process is successful, a message will be displayed, indicating that the "Display driver stopped responding and has recovered."

For more information, see Timeout Detection and Recovery (TDR), TDR Registry Keys and TDR changes in Windows 8 which are located in Debugging Tips for the Windows Display Driver Model (WDDM).

Resolution

The GPU is taking more time than permitted to display graphics to your monitor. This behavior can occur for one or more of the following reasons:

- You may need to install the latest updates for your display driver, so that it properly supports the TDR process.
- Hardware issues that impact the ability of the video card to operate properly, including:
  - Over-clocked components, such as the motherboard
  - Incorrect component compatibility and settings (especially memory configuration and timings)
  - Insufficient system cooling
  - Insufficient system power
  - Defective parts (memory modules, motherboards, etc.)
- Visual effects, or too many programs running in the background may be slowing your PC down so that the video card can not respond as necessary.

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause.

```
1: kd> !analyze -v
*******************************************************************************
*                                                                             *
*                        Bugcheck Analysis                                    *
*                                                                             *
*******************************************************************************

VIDEO_TDR_FAILURE (116)
Attempt to reset the display driver and recover from timeout failed.
Arguments:
Arg1: ffffe000c2c404c0, Optional pointer to internal TDR recovery context (TDR_RECOVERY_CONTEXT).
Arg2: fffff8016470c14c, The pointer into responsible device driver module (e.g. owner tag).
Arg3: ffffffffc000009a, Optional error code (NTSTATUS) of the last failed operation.
Arg4: 0000000000000004, Optional internal context dependent data.

...
```

Also displayed will be the faulting module name

```
MODULE_NAME: nvlddmkm

IMAGE_NAME:  nvlddmkm.sys
```

You can use the **lm (List Loaded Modules)**command to display information about the faulting driver, including the timestamp.

```
1: kd> lmvm nvlddmkm
Browse full module list
start             end                 module name
fffff801`63ec0000 fffff801`649a7000   nvlddmkm T (no symbols)
    Loaded symbol image file: nvlddmkm.sys
    Image path: \SystemRoot\system32\DRIVERS\nvlddmkm.sys
    Image name: nvlddmkm.sys
    Browse all global symbols  functions  data
    Timestamp:        Wed Jul  8 15:43:44 2015 (559DA7A0)
    CheckSum:         00AA7491
    ImageSize:        00AE7000
    Translations:     0000.04b0 0000.04e4 0409.04b0 0409.04e4
```

Parameter 1 contains a pointer to the TDR_RECOVERY_CONTEXT. As shown in the !analyze output, if you have symbols for the associated code, you can use the dt command to display this data.

```
1: kd> dt dxgkrnl!_TDR_RECOVERY_CONTEXT ffffe000c2c404c0
   +0x000 Signature        : 0x52445476
   +0x008 pState           : 0xffffe000`c2b12a40 ??
   +0x010 TimeoutReason    : 9 ( TdrEngineTimeoutPromotedToAdapterReset )
   +0x018 Tick             : _ULARGE_INTEGER 0xb2
   +0x020 pAdapter         : 0xffffe000`c2a89010 DXGADAPTER
   +0x028 pVidSchContext   : (null)
   +0x030 GPUTimeoutData   : _TDR_RECOVERY_GPU_DATA
   +0x048 CrtcTimeoutData  : _TDR_RECOVERY_CONTEXT::<unnamed-type-CrtcTimeoutData>
   +0x050 pProcessName     : (null)
   +0x058 DbgOwnerTag      : 0xfffff801`6470c14c
   +0x060 PrivateDbgInfo   : _TDR_DEBUG_REPORT_PRIVATE_INFO
   +0xb00 pDbgReport       : 0xffffe000`c2c3f750 _WD_DEBUG_REPORT
   +0xb08 pDbgBuffer       : 0xfffffc000`bd000000 Void
   +0xb10 DbgBufferSize    : 0x37515
   +0xb18 pDumpBufferHelper : (null)
   +0xb20 pDbgInfoExtension : 0xfffffc000`ba7e47a0 _DXGKARG_COLLECTDBGINFO_EXT
   +0xb28 pDbgBufferUpdatePrivateInfo : 0xfffffc000`bd000140 Void
   +0xb30 ReferenceCount   : 0n1
   +0xb38 pResetCompletedEvent : (null)
```

Parameter 2 contains a pointer into the responsible device driver module (for example, the owner tag).

```
1: kd> ub fffff8016470c14c
nvlddmkm+0x84c132:
fffff801`6470c132 cc              int     3
fffff801`6470c133 cc              int     3
fffff801`6470c134 48ff254d2deaff  jmp     qword ptr [nvlddmkm+0x6eee88 (fffff801`645aee88)]
fffff801`6470c13b cc              int     3
fffff801`6470c13c 48ff252d2eeaff  jmp     qword ptr [nvlddmkm+0x6eef70 (fffff801`645aef70)]
fffff801`6470c143 cc              int     3
fffff801`6470c144 48ff257d2deaff  jmp     qword ptr [nvlddmkm+0x6eeec8 (fffff801`645aeec8)]
fffff801`6470c14b cc              int     3
```

You may wish to examine the stack trace using the **k, kb, kc, kd, kp, kP, kv (Display Stack Backtrace)** command.

```
1: kd> k
 # Child-SP          RetAddr           Call Site
00 ffffd001`7d53d918 fffff801`61ba2b4c nt!KeBugCheckEx [d:\th\minkernel\ntos\ke\amd64\procstat.asm @ 122]
01 ffffd001`7d53d920 fffff801`61b8da0e dxgkrnl!TdrBugcheckOnTimeout+0xec
[d:\th\windows\core\dxkernel\dxgkrnl\core\dxgtdr.cxx @ 2731]
02 ffffd001`7d53d960 fffff801`61b8dd7f dxgkrnl!ADAPTER_RENDER::Reset+0x15e
[d:\th\windows\core\dxkernel\dxgkrnl\core\adapter.cxx @ 19443]
03 ffffd001`7d53d990 fffff801`61ba2385 dxgkrnl!DXGADAPTER::Reset+0x177
[d:\th\windows\core\dxkernel\dxgkrnl\core\adapter.cxx @ 19316]
04 ffffd001`7d53d9e0 fffff801`63c5fba7 dxgkrnl!TdrResetFromTimeout+0x15
[d:\th\windows\core\dxkernel\dxgkrnl\core\dxgtdr.cxx @ 2554]
05 ffffd001`7d53da10 fffff801`63c47e5d dxgmms1!VidSchiRecoverFromTDR+0x11b
[d:\th\windows\core\dxkernel\dxgkrnl\dxgmms1\vidsch\vidscher.cxx @ 1055]
06 ffffd001`7d53dbc0 fffff801`aa55c698 dxgmms1!VidSchiWorkerThread+0x8d
[d:\th\windows\core\dxkernel\dxgkrnl\dxgmms1\vidsch\vidschi.cxx @ 426]
07 ffffd001`7d53dc00 fffff801`aa5c9306 nt!PspSystemThreadStartup+0x58 [d:\th\minkernel\ntos\ps\psexec.c @
6845]
08 ffffd001`7d53dc60 00000000`00000000 nt!KxStartSystemThread+0x16 [d:\th\minkernel\ntos\ke\amd64\threadbg.asm
@ 80]
```

You can also set a breakpoint in the code leading up to this stop code and attempt to single step forward into the faulting code, if you can consistently reproduce the stop code.

For more information see the following topics:

Crash dump analysis using the Windows debuggers (WinDbg)

If you are not equipped to use the Windows debugger to work on this problem, you can use some basic troubleshooting techniques.

- Check the System Log in Event Viewer for additional error messages that might help identify the device or driver that is causing this bug check.
- If a driver is identified in the bug check message, disable the driver or check with the manufacturer for driver updates.
- Verify that all graphics related software such as DirectX and OpenGL are up to date, and any graphics intensive applications (such as games) are fully patched.
- Confirm that any new hardware that is installed is compatible with the installed version of Windows. For example, you can get information about required hardware at Windows 10 Specifications.
- **Using Safe Mode**

  Consider using Safe Mode to help isolate this issue. Using Safe Mode loads only the minimum required drivers and system services during the Windows startup. To enter Safe Mode, use **Update and Security** in Settings. Select **Recovery**->**Advanced startup** to boot to maintenance mode. At the resulting menu, choose **Troubleshoot**-> **Advanced Options** -> **Startup Settings** -> **Restart**. After Windows restarts to the **Startup Settings** screen, select option, 4, 5 or 6 to boot to Safe Mode.

  Safe Mode may be available by pressing a function key on boot, for example F8. Refer to information from the manufacturer for specific startup options.
- Run the Windows Memory Diagnostics tool, to test the memory. In the control panel search box, type Memory, and then click **Diagnose your computer's memory problems**. After the test is run, use Event viewer to view the results under the System log. Look for the *MemoryDiagnostics-Results* entry to view the results.
- You can try running the hardware diagnostics supplied by the system manufacturer.
- For additional general troubleshooting information, see **Blue Screen Data**.

Remarks

Hardware certification requirements

For information on requirements that hardware devices must meet when they implement TDR, refer to the WHCK documentation on *Device.Graphics...TDRResiliency*.

Bug Check 0x117: VIDEO_TDR_TIMEOUT_DETECTED

10/23/2019 • 5 minutes to read • Edit Online

The VIDEO_TDR_TIMEOUT_DETECTED bug check has a value of 0x00000117. This indicates that the display driver failed to respond in a timely fashion.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

VIDEO_TDR_TIMEOUT_DETECTED Parameters

PARAMETER	DESCRIPTION
1	The pointer to the internal TDR recovery context, if available.
2	A pointer into the responsible device driver module (for example, the owner tag).
3	The secondary driver-specific bucketing key.
4	Internal context dependent data, if available.

Cause

A common stability problem in graphics occurs when the system appears completely frozen or hung while processing an end-user command or operation. Usually the GPU is busy processing intensive graphics operations, typically during game-play. No screen updates occur, and users assume that their system is frozen. Users usually wait a few seconds and then reboot the system by pressing the power button. Windows tries to detect these problematic hang situations and dynamically recover a responsive desktop.

This process of detection and recovery is known as Timeout Detection and Recovery (TDR). The default timeout is 2 seconds. In the TDR process for video cards, the operating system's GPU scheduler calls the display miniport driver's *DxgkDdiResetFromTimeout* function to reinitialize the driver and reset the GPU.

If the recovery process is successful, a message will be displayed, indicating that the "Display driver stopped responding and has recovered."

For more information, see Timeout Detection and Recovery (TDR), TDR Registry Keys and TDR changes in Windows 8 which are located in Debugging Tips for the Windows Display Driver Model (WDDM).

Resolution

The GPU is taking more time than permitted to display graphics to your monitor. This behavior can occur for one or more of the following reasons:

- You may need to install the latest updates for your display driver, so that it properly supports the TDR process.
- Hardware issues that impact the ability of the video card to operate properly, including:
  - Over-clocked components, such as the motherboard
  - Incorrect component compatibility and settings (especially memory configuration and timings)
  - Insufficient system cooling
  - Insufficient system power
  - Defective parts (memory modules, motherboards, etc.)
- Visual effects, or too many programs running in the background may be slowing your PC down so that the video card can not respond as necessary.

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause.

```
3: kd> !analyze -v
*******************************************************************************
*                                                                             *
*                        Bugcheck Analysis                                    *
*                                                                             *
*******************************************************************************

VIDEO_TDR_TIMEOUT_DETECTED (117)
The display driver failed to respond in timely fashion.
(This code can never be used for a real bugcheck.)
Arguments:
Arg1: 8975d500, Optional pointer to internal TDR recovery context (TDR_RECOVERY_CONTEXT).
Arg2: 9a02381e, The pointer into responsible device driver module (e.g owner tag).
Arg3: 00000000, The secondary driver specific bucketing key.
Arg4: 00000000, Optional internal context dependent data.

...
```

Also displayed will be the faulting module name

```
MODULE_NAME: atikmpag

IMAGE_NAME:  atikmpag.sys
```

You can use the lmv command to display information about the faulting driver, including the timestamp.

```
3: kd> lmvm atikmpag
Browse full module list
start    end        module name
9a01a000 9a09a000   atikmpag T (no symbols)
    Loaded symbol image file: atikmpag.sys
    Image path: atikmpag.sys
    Image name: atikmpag.sys
    Browse all global symbols  functions  data
    Timestamp:        Fri Dec  6 12:20:32 2013 (52A23190)
    CheckSum:         0007E58A
    ImageSize:        00080000
    Translations:     0000.04b0 0000.04e4 0409.04b0 0409.04e4
```

Parameter 1 contains a pointer to the TDR_RECOVERY_CONTEXT.

```
3: kd> dt dxgkrnl!_TDR_RECOVERY_CONTEXT fffffa8010041010
   +0x000 Signature        : ??
   +0x004 pState           : ????
   +0x008 TimeoutReason    : ??
   +0x010 Tick             : _ULARGE_INTEGER
   +0x018 pAdapter         : ????
   +0x01c pVidSchContext   : ????
   +0x020 GPUTimeoutData   : _TDR_RECOVERY_GPU_DATA
   +0x038 CrtcTimeoutData  : _TDR_RECOVERY_CONTEXT::<unnamed-type-CrtcTimeoutData>
   +0x040 DbgOwnerTag      : ??
   +0x048 PrivateDbgInfo   : _TDR_DEBUG_REPORT_PRIVATE_INFO
   +0xae0 pDbgReport       : ????
   +0xae4 pDbgBuffer       : ????
   +0xae8 DbgBufferSize    : ??
   +0xaec pDumpBufferHelper : ????
   +0xaf0 pDbgInfoExtension : ????
   +0xaf4 pDbgBufferUpdatePrivateInfo : ????
   +0xaf8 ReferenceCount   : ??
Memory read error 10041b08
```

Parameter 2 contains a pointer into the responsible device driver module (for example, the owner tag).

```
BUGCHECK_P2: ffffffff9a02381e
```

You may wish to examine the stack trace using the **k, kb, kc, kd, kp, kP, kv (Display Stack Backtrace)** command.

```
3: kd> k
 # ChildEBP RetAddr
00 81d9ace0 976e605e dxgkrnl!TdrUpdateDbgReport+0x93
[d:\blue_gdr\windows\core\dxkernel\dxgkrnl\core\dxgtdr.cxx @ 944]
01 81d9acfc 976ddead dxgkrnl!TdrCollectDbgInfoStage2+0x195
[d:\blue_gdr\windows\core\dxkernel\dxgkrnl\core\dxgtdr.cxx @ 1759]
02 81d9ad24 976e664f dxgkrnl!DXGADAPTER::Reset+0x23f
[d:\blue_gdr\windows\core\dxkernel\dxgkrnl\core\adapter.cxx @ 14972]
03 81d9ad3c 977be9e0 dxgkrnl!TdrResetFromTimeout+0x16
[d:\blue_gdr\windows\core\dxkernel\dxgkrnl\core\dxgtdr.cxx @ 2465]
04 81d9ad50 977b7518 dxgmms1!VidSchiRecoverFromTDR+0x13
[d:\blue_gdr\windows\core\dxkernel\dxgkrnl\dxgmms1\vidsch\vidscher.cxx @ 1018]
05 (Inline) -------- dxgmms1!VidSchiRun_PriorityTable+0xfa71
06 81d9ad70 812c01d4 dxgmms1!VidSchiWorkerThread+0xfaf2
[d:\blue_gdr\windows\core\dxkernel\dxgkrnl\dxgmms1\vidsch\vidschi.cxx @ 424]
07 81d9adb0 81325fb1 nt!PspSystemThreadStartup+0x58 [d:\blue_gdr\minkernel\ntos\ps\psexec.c @ 5884]
08 81d9adbc 00000000 nt!KiThreadStartup+0x15 [d:\blue_gdr\minkernel\ntos\ke\i386\threadbg.asm @ 81]
```

You can also set a breakpoint in the code leading up to this stop code and attempt to single step forward into the faulting code, if you can consistently reproduce the stop code.

For more information see the following topics:

Crash dump analysis using the Windows debuggers (WinDbg)

If you are not equipped to use the Windows debugger to work on this problem, you can use some basic troubleshooting techniques.

- Check the System Log in Event Viewer for additional error messages that might help identify the device or driver that is causing this bug check.
- If a driver is identified in the bug check message, disable the driver or check with the manufacturer for driver updates.
- Verify that all graphics related software such as DirectX and OpenGL are up to date, and any graphics

intensive applications (such as games) are fully patched.

- Confirm that any new hardware that is installed is compatible with the installed version of Windows. For example, you can get information about required hardware at Windows 10 Specifications.

- **Using Safe Mode**

  Consider using Safe Mode to help isolate this issue. Using Safe Mode loads only the minimum required drivers and system services during the Windows startup. To enter Safe Mode, use **Update and Security** in Settings. Select **Recovery**->**Advanced startup** to boot to maintenance mode. At the resulting menu, choose **Troubleshoot**-> **Advanced Options** -> **Startup Settings** -> **Restart**. After Windows restarts to the **Startup Settings** screen, select option, 4, 5 or 6 to boot to Safe Mode.

  Safe Mode may be available by pressing a function key on boot, for example F8. Refer to information from the manufacturer for specific startup options.

- Run the Windows Memory Diagnostics tool, to test the memory. In the control panel search box, type Memory, and then click **Diagnose your computer's memory problems**. After the test is run, use Event viewer to view the results under the System log. Look for the *MemoryDiagnostics-Results* entry to view the results.

- You can try running the hardware diagnostics supplied by the system manufacturer.

- For additional general troubleshooting information, see **Blue Screen Data**.

Remarks

Hardware certification requirements

For information on requirements that hardware devices must meet when they implement TDR, refer to the WHCK documentation on *Device.Graphics...TDRResiliency*.

Bug Check 0x119: VIDEO_SCHEDULER_INTERNAL_ERROR

9/11/2019 • 2 minutes to read • Edit Online

The VIDEO_SCHEDULER_INTERNAL_ERROR bug check has a value of 0x00000119. This indicates that the video scheduler has detected a fatal violation.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

VIDEO_SCHEDULER_INTERNAL_ERROR Parameters

Parameter 1 is the only parameter of interest and identifies the exact violation.

PARAMETER 1	CAUSE OF ERROR
0x1	The driver has reported an invalid fence ID.
0x2	The driver failed upon the submission of a command.
0x3	The driver failed upon patching the command buffer.
0x4	The driver reported an invalid flip capability.

Resolution

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause.

Bug Check 0x11A: EM_INITIALIZATION_FAILURE

7/2/2019 • 2 minutes to read • Edit Online

The EM_INITIALIZATION_FAILURE bug check has a value of 0x0000011A.

This bug check appears very infrequently.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Bug Check 0x11B: DRIVER_RETURNED_HOLDING_CANCEL_LOCK

7/2/2019 • 2 minutes to read • Edit Online

The DRIVER_RETURNED_HOLDING_CANCEL_LOCK bug check has a value of 0x0000011B. This bug check indicates that a driver has returned from a *cancel* routine that holds the global cancel lock. This causes all later cancellation calls to fail, and results in either a deadlock or another bug check.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

DRIVER_RETURNED_HOLDING_CANCEL_LOCK Parameters

PARAMETER	DESCRIPTION
1	The address of the IRP that was canceled (might not be valid).
2	The address of the *cancel* routine.

Remarks

The cancel spin lock should have been released by the *cancel* routine.

The driver calls the IoCancelIrpIoCancelIrp function to cancel an individual I/O request packet (IRP). This function acquires the cancel spin lock, sets the cancel flag in the IRP, and then calls the *cancel* routine specified by the appropriate field in the IRP, if a routine was specified. The *cancel* routine is expected to release the cancel spin lock. If there is no *cancel* routine, the cancel spin lock is released.

Bug Check 0x11C: ATTEMPTED_WRITE_TO_CM_PROTECTED_STORAGE

7/2/2019 • 2 minutes to read • Edit Online

The ATTEMPTED_WRITE_TO_CM_PROTECTED_STORAGE bug check has a value of 0x0000011C. This bug check indicates that an attempt was made to write to the read-only protected storage of the configuration manager.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

ATTEMPTED_WRITE_TO_CM_PROTECTED_STORAGE Parameters

PARAMETER	DESCRIPTION
1	Virtual address for the attempted write
2	PTE contents
3	Reserved
4	Reserved

Remarks

When it is possible, the name of the driver that is attempting the write operation is printed as a Unicode string on the bug check screen and then saved in KiBugCheckDriver.

Bug Check 0x11D: EVENT_TRACING_FATAL_ERROR

7/2/2019 • 2 minutes to read • Edit Online

The EVENT_TRACING_FATAL_ERROR bug check has a value of 0x0000011D. This bug check indicates that the Event Tracing subsystem has encountered an unexpected fatal error.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Bug Check 0x11E: TOO_MANY_RECURSIVE_FAULTS

7/2/2019 • 2 minutes to read • Edit Online

The TOO_MANY_RECURSIVE_FAULTS bug check has a value of 0x0000011E. This indicates that a file system has caused too many recursive faults under low resource conditions to be handled.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

TOO_MANY_RECURSIVE_FAULTS Parameters

None

Bug Check 0x11F: INVALID_DRIVER_HANDLE

7/2/2019 • 2 minutes to read • Edit Online

The INVALID_DRIVER_HANDLE bug check has a value of 0x0000011F. This indicates that someone has closed the initial handle for a driver between inserting the driver object and referencing the handle.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

INVALID_DRIVER_HANDLE Parameters

PARAMETER	DESCRIPTION
1	The handle value for the driver object.
2	The status returned trying to reference the object.
3	The address of the PDRIVER_OBJECT.
4	Reserved

Bug Check 0x120: BITLOCKER_FATAL_ERROR

7/2/2019 • 2 minutes to read • Edit Online

The BITLOCKER_FATAL_ERROR bug check has a value of 0x00000120. This indicates that BitLocker drive encryption encountered a problem that it cannot recover from.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

BITLOCKER_FATAL_ERROR Parameters

PARAMETER	DESCRIPTION
1	Type of problem
2	Reserved
3	Reserved
4	Reserved

Bug Check 0x121: DRIVER_VIOLATION

7/2/2019 • 2 minutes to read • Edit Online

The DRIVER_VIOLATION bug check has a value of 0x00000121. This bug check indicates that a driver has caused a violation.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

DRIVER_VIOLATION Parameters

PARAMETER	DESCRIPTION
1	Describes the type of violation
2	Reserved
3	Reserved

Remarks

Use a kernel debugger and view the call stack to determine the name of the driver that caused the violation: the **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause, then enter one of the **k (Display Stack Backtrace)** commands to view the call stack.

Bug Check 0x122: WHEA_INTERNAL_ERROR

7/2/2019 • 2 minutes to read • Edit Online

The WHEA_INTERNAL_ERROR bug check has a value of 0x00000122. This bug check indicates that an internal error in the Windows Hardware Error Architecture (WHEA) has occurred. Errors can result from a bug in the implementation of a platform-specific hardware error driver (PSHED) plug-in supplied by a vendor, the firmware implementation of error records, or the firmware implementation of error injection.

IMPORTANT

This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

WHEA_INTERNAL_ERROR Parameters

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x1	Size of memory	Error source count	0	Failed to allocate enough memory for all the error sources in the hardware error source table.
0x2	Number of processors	0	0	Failed to allocate enough memory for a WHEA information block for each processor.
0x5	Status	Phase (The initialization phase for the bug check)	0	WHEA failed to allocate enough memory for the error sources, or the error source enumeration failed.
0x6	Status	Phase	Error source type	Failed to initialize the error source (Parameter 4) during the phase specified by Parameter 3.
0x7	Status	0	0	Failed to allocate enough memory.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x8	Number of error sources	0	0	Failed to allocate enough memory for all the error source descriptors.
0x9	Error source type	Source ID	0	WHEA received an uncorrected error source from an invalid error source.
0xA	Error source type	Source ID	0	Failed to allocate an error record for an uncorrected error.
0xB	Error source type	Source ID	0	Failed to populate the error record for an uncorrected error.

If Parameter 1 is equal to 0x6, 0x9, 0xA, or 0xB, one of the other parameters contains the error source type. The following table gives possible values for the error source type.

VALUE	DESCRIPTION
0x00	Machine check exception
0x01	Corrected machine check
0x02	Corrected platform error
0x03	Non-maskable interrupt
0x04	PCI express error
0x05	Other types of error sources/Generic
0x06	IA64 INIT error source
0x07	BOOT error source
0x08	SCI-based generic error source
0x09	Itanium machine check abort
0x0A	Itanium machine check
0x0B	Itanium corrected platform error

Bug Check 0x123: CRYPTO_SELF_TEST_FAILURE

7/2/2019 • 2 minutes to read • Edit Online

The CRYPTO_SELF_TEST_FAILURE bug check has a value of 0x00000123. This indicates that the cryptographic subsystem failed a mandatory algorithm self-test during bootstrap.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

CRYPTO_SELF_TEST_FAILURE Parameters

None

Bug Check 0x124: WHEA_UNCORRECTABLE_ERROR

10/23/2019 • 3 minutes to read • Edit Online

The WHEA_UNCORRECTABLE_ERROR bug check has a value of 0x00000124. This bug check indicates that a fatal hardware error has occurred. This bug check uses the error data that is provided by the Windows Hardware Error Architecture (WHEA).

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

WHEA_UNCORRECTABLE_ERROR Parameters

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x0	Address of WHEA_ERROR_RECORD structure.	High 32 bits of MCi_STATUS MSR for the MCA bank that had the error.	Low 32 bits of MCi_STATUS MSR for the MCA bank that had the error.	A machine check exception occurred. These parameter descriptions apply if the processor is based on the x64 architecture, or the x86 architecture that has the MCA feature available (for example, Intel Pentium Pro, Pentium IV, or Xeon).
0x1	Address of WHEA_ERROR_RECORD structure.	Reserved.	Reserved.	A corrected machine check exception occurred.
0x2	Address of WHEA_ERROR_RECORD structure.	Reserved.	Reserved.	A corrected platform error occurred.
0x3	Address of WHEA_ERROR_RECORD structure.	Reserved.	Reserved.	A nonmaskable Interrupt (NMI) error occurred.
0x4	Address of WHEA_ERROR_RECORD structure.	Reserved	Reserved.	An uncorrectable PCI Express error occurred.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x5	Address of WHEA_ERROR_RECORD structure.	Reserved.	Reserved.	A generic hardware error occurred.
0x6	Address of WHEA_ERROR_RECORD structure	Reserved.	Reserved.	An initialization error occurred.
0x7	Address of WHEA_ERROR_RECORD structure.	Reserved.	Reserved.	A BOOT error occurred.
0x8	Address of WHEA_ERROR_RECORD structure	Reserved.	Reserved.	A Scalable Coherent Interface (SCI) generic error occurred.
0x9	Address of WHEA_ERROR_RECORD structure.	Length, in bytes, of the SAL log.	Address of the SAL log.	An uncorrectable Itanium-based machine check abort error occurred.
0xA	Address of WHEA_ERROR_RECORD structure	Reserved.	Reserved.	A corrected Itanium-based machine check error occurred.
0xB	Address of WHEA_ERROR_RECORD structure.	Reserved.	Reserved.	A corrected Itanium platform error occurred.

Cause

This bug check is typically related to physical hardware failures. It can be heat related, defective hardware, memory or even a processor that is beginning to fail or has failed. If over-clocking has been enabled, try disabling it. Confirm that any cooling systems such as fans are functional. Run system diagnostics to confirm that the system memory is not defective. It is less likely, but possible that a driver is causing the hardware to fail with this bug check.

For additional general bug check troubleshooting information, see **Blue Screen Data**.

Remarks

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause.

Parameter 1 identifies the type of error source that reported the error. Parameter 2 holds the address of the WHEA_ERROR_RECORD structure that describes the error condition.

When a hardware error occurs, WHEA creates an error record to store the error information associated with the hardware error condition. Each error record is described by a WHEA_ERROR_RECORD structure. The Windows kernel includes the error record with the Event Tracing for Windows (ETW) hardware error event that it raises in response to the error so that the error record is saved in the system event log. The format of the error records that are used by WHEA are based on the Common Platform Error Record as described in Appendix N of version 2.2 of the Unified Extensible Firmware Interface (UEFI) Specification. For more information, see WHEA_ERROR_RECORD and Windows Hardware Error Architecture (WHEA).

You can use **!errrec** <addr> to display the WHEA_ERROR_RECORD structure using the address provided in Parameter 2. The **!whea** and **!errpkt** extensions can be used to display additional WHEA information.

For more information see the following topics:

Crash dump analysis using the Windows debuggers (WinDbg)

Analyzing a Kernel-Mode Dump File with WinDbg

Using the !analyze Extension and !analyze

This bug check is not supported in Windows versions prior to Windows Vista. Instead, machine check exceptions are reported through **bug check 0x9C**.

Bug Check 0x125: NMR_INVALID_STATE

7/2/2019 • 2 minutes to read • Edit Online

The NMR_INVALID_STATE bug check has a value of 0x00000125. This indicates that NMR (network module registrar) has detected an invalid state. See parameter 1 for the state type.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

NMR_INVALID_STATE Parameters

PARAMETER	DESCRIPTION

PARAMETER	DESCRIPTION
1	The subtype of the bugcheck. 0x0 : Machine Check Exception Parameter 2 - Address of the WHEA_ERROR_RECORD structure. Parameter 3 - High order 32-bits of the MCi_STATUS value. Parameter 4 - Low order 32-bits of the MCi_STATUS value. 0x1 : Corrected Machine Check Parameter 2 - Address of the WHEA_ERROR_RECORD structure. 0x2 : Corrected Platform Error Parameter 2 - Address of the WHEA_ERROR_RECORD structure. 0x3 : Non-maskable Interrupt Parameter 2 - Address of the WHEA_ERROR_RECORD structure. 0x4 : PCI Express Error Parameter 2 - Address of the WHEA_ERROR_RECORD structure. 0x5 : Generic Error Parameter 2 - Address of the WHEA_ERROR_RECORD structure. 0x6 : INIT Error Parameter 2 - Address of the WHEA_ERROR_RECORD structure. 0x7 : BOOT Error Parameter 2 - Address of the WHEA_ERROR_RECORD structure. 0x8 : SCI Generic Error Parameter 2 - Address of the WHEA_ERROR_RECORD structure. 0x9 : Itanium Machine Check Abort Parameter 2 - Address of the WHEA_ERROR_RECORD structure. Parameter 3 - Length in bytes of the SAL log. Parameter 4 - Address of the SAL log. 0xa : Itanium Corrected Machine Check Parameter 2 - Address of the WHEA_ERROR_RECORD structure. 0xb : Itanium Corrected Platform Error Parameter 2 - Address of the WHEA_ERROR_RECORD structure.
2	Pointer to the NMI Handle
3	Pointer to the expected type, when available
4	Reserved

Bug Check 0x126: NETIO_INVALID_POOL_CALLER

7/2/2019 • 2 minutes to read • Edit Online

The NETIO_INVALID_POOL_CALLER bug check has a value of 0x00000126. This indicates that an invalid pool request has been made to netio managed memory pool, e.g. FSB and MDL.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

NETIO_INVALID_POOL_CALLER Parameters

PARAMETER	DESCRIPTION
1	The subtype of the bugcheck. 0x1 : Invalid pool. Pool is at an invalid state. Parameter 2 - Pointer to memory block or MDL. Parameter 3 - Pointer to page. Parameter 4 - Pointer to CPU pool. 0x2 : Invalid MDL. MDL is at an invalid state. Parameter 2 - Pointer to MDL. Parameter 3 - Pointer to CPU pool. Parameter 4 - Pointer to pool header.
2	See parameter 1
3	See parameter 1
4	See parameter 1

Bug Check 0x127: PAGE_NOT_ZERO

7/2/2019 • 2 minutes to read • Edit Online

The PAGE_NOT_ZERO bug check has a value of 0x00000127. This bug check indicates that a page that should have been filled with zeros was not. This bug check might occur because of a hardware error or because a privileged component of the operating system modified a page after freeing it.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

PAGE_NOT_ZERO Parameters

PARAMETER	DESCRIPTION
1	Virtual address that maps the corrupted page
2	Physical page number
3	Zero (Reserved)
4	Zero (Reserved)

Bug Check 0x128: WORKER_THREAD_RETURNED_WITH_BAD_IO_PRIORITY

7/2/2019 • 2 minutes to read • Edit Online

The WORKER_THREAD_RETURNED_WITH_BAD_IO_PRIORITY bug check has a value of 0x00000128. This indicates that a worker threads IOPriority was wrongly modified by the called worker routine.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

WORKER_THREAD_RETURNED_WITH_BAD_IO_PRIORITY Parameters

PARAMETER	DESCRIPTION
1	Address of worker routine (Use the **ln (List Nearest Symbols)** command on this address to find the offending driver)
2	Current IoPrioirity value
3	Workitem parameter
4	Workitem address

Bug Check 0x129: WORKER_THREAD_RETURNED_WITH_BAD_PAGING_IO_PRIORITY

7/2/2019 • 2 minutes to read • Edit Online

The WORKER_THREAD_RETURNED_WITH_BAD_PAGING_IO_PRIORITY bug check has a value of 0x00000129. This indicates that a worker threads Paging IOPriority was wrongly modified by the called worker routine.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

WORKER_THREAD_RETURNED_WITH_BAD_PAGING_IO_PRIORITY Parameters

PARAMETER	DESCRIPTION
1	Address of worker routine Use the **ln (List Nearest Symbols)** command on this address to find the offending driver.
2	Current Paging IoPrioirity value
3	Workitem parameter
4	Workitem address

Bug Check 0x12A: MUI_NO_VALID_SYSTEM_LANGUAGE

7/2/2019 • 2 minutes to read • Edit Online

The MUI_NO_VALID_SYSTEM_LANGUAGE bug check has a value of 0x0000012A. This indicates that Windows did not find any installed, licensed language packs for the system default UI language.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

MUI_NO_VALID_SYSTEM_LANGUAGE Parameters

PARAMETER	DESCRIPTION
1	The subtype of the bugcheck 0x1 : Windows did not find any installed language packs during phase I initialization. Parameter 2 - NT status code that describes the reason of failure. 0x2 : Windows did not find any installed, licensed language packs for the system default UI language during kernel cache creation. Parameter 2 - NT status code that describes the reason of failure. .
2	See parameter 1
3	Reserved
4	Reserved

Bug Check 0x12B: FAULTY_HARDWARE_CORRUPTED_PAGE

7/13/2019 • 2 minutes to read • Edit Online

The FAULTY_HARDWARE_CORRUPTED_PAGE bug check has a value of 0x0000012B. This bug check indicates that the Windows memory manager detected corruption, and the corruption could only have been caused by a component accessing memory using physical addressing.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

FAULTY_HARDWARE_CORRUPTED_PAGE Parameters

There are two scenarios where the Memory Manager will raise FAULTY_HARDWARE_CORRUPTED_PAGE bug checks, with two different sets of parameters.

If parameters 3 and 4 are both zero, the bug check indicates that Memory Manager detected a single-bit error on a page that was expected to be zeroed.

If parameters 3 and 4 are non-zero, the bug check is raised by the Compressed Store Manager due to a failure to decompress a page due to physical memory corruption.

Memory Manager Page Not Zero Error Parameters

This bug check indicates that a single-bit error was found in this page. This is a hardware memory error.

PARAMETER	DESCRIPTION
1	Virtual address maps to the corrupted page
2	Physical page number
3	Zero
4	Zero

Compressed Store Manager Error Parameters

This bug check indicates that a store manager memory error has occurred. It may be an authentication failure, a CRC failure, or a decompression failure.

PARAMETER	DESCRIPTION
1	FailStatus - Indicates the type of failure

PARAMETER	DESCRIPTION
2	The CompressedSize of the page that is being read
3	Source Buffer
4	Target Buffer

Cause

This bugcheck can only occur by memory corruption due to physical memory access. The causes for physical memory corruption include:

1. Defective RAM hardware
2. A driver or device incorrectly modifying physical pages via an incorrect DMA operation or associated MDL.
3. Corruption caused by a hardware device or firmware corrupting memory, such as firmware illegally modifying physical pages across a power transition.

NOTE: Compressed Store Manager can detect if the corruption was caused by a single-bit error, and automatically corrects this condition without raising a bug check. This bugcheck is reported by the Compressed Store Manager if the corruption was not caused by a single bit error.

For more information on Windows memory manager and memory compression, see Windows Internals 7th Edition Part 1 by Pavel Yosifovich, Mark E. Russinovich, David A. Solomon and Alex Ionescu.

Resolution

Windows Memory Diagnostics Tool

To investigate if this bug check is caused by defective RAM hardware, run the Windows Memory Diagnostics tool. In the control panel search box, type Memory, and then click *Diagnose your computer's memory problems*. After the test is run, use Event viewer to view the results under the System log. Look for the *MemoryDiagnostics-Results* entry to view the results.

See Also

Bug Check Code Reference

Windows Kernel-Mode Memory Manager

Channel 9 video on memory compression

Bug Check 0x12C: EXFAT_FILE_SYSTEM

7/2/2019 • 2 minutes to read • Edit Online

The EXFAT_FILE_SYSTEM bug check has a value of 0x0000012C. This bug check indicates that a problem occurred in the Extended File Allocation Table (exFAT) file system.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

EXFAT_FILE_SYSTEM Parameters

PARAMETER	DESCRIPTION
1	Specifies source file and line number information. The high 16 bits (the first four hexadecimal digits after the "0x") determine the source file by its identifier number. The low 16 bits determine the source line in the file where the bug check occurred.
2	If **FppExceptionFilter** is on the stack, this parameter specifies the address of the exception record.
3	If **FppExceptionFilter** is on the stack, this parameter specifies the address of the context record.
4	Reserved.

Cause

This bug check is caused by the file system as a last resort when its internal accounting is in an unsupportable state and to continue poses a large risk of data loss. The file system never causes this bug check when the on disk structures are corrupted, the disk sectors go bad, or a memory allocation fails. Bad sectors could lead to a bug check, for example, when a page fault occurs in kernel code or data and the memory manager cannot read the pages. However, for this bug check, the file system is not the cause.

Resolution

To debug this problem: Use the **.cxr (Display Context Record)** command together with Parameter 3, and then use **kb (Display Stack Backtrace)**.

Bug Check 0x12D: VOLSNAP_OVERLAPPED_TABLE_ACCESS

7/2/2019 • 2 minutes to read • Edit Online

The VOLSNAP_OVERLAPPED_TABLE_ACCESS bug check has a value of 0x0000012D. This indicates that volsnap tried to access a common table from two different threads which may result in table corruption and eventually corrupt the table.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

VOLSNAP_OVERLAPPED_TABLE_ACCESS Parameters

None

Bug Check 0x12E: INVALID_MDL_RANGE

7/2/2019 • 2 minutes to read • Edit Online

The INVALID_MDL_RANGE bug check has a value of 0x0000012E. This indicates that a driver has called the IoBuildPartialMdl() function and passed it an MDL to map part of a source MDL, but the virtual address range specified is outside the range in the source MDL. This is typically a driver bug.

The source and target MDLs, as well as the address range length to be mapped are the arguments to the IoBuildPartialMdl() function, i.e.;) .

```
IoBuildPartialMdl(
        IN PMDL SourceMdl,
        IN OUT PMDL TargetMdl,
        IN PVOID VirtualAddress,
        IN ULONG Length
```

IMPORTANT

This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

INVALID_MDL_RANGE Parameters

PARAMETER	DESCRIPTION
1	SourceMdl
2	TargetMdl
3	VirtualAddress
4	Length

Bug Check 0x12F: VHD_BOOT_INITIALIZATION_FAILED

7/2/2019 • 2 minutes to read • Edit Online

The VHD_BOOT_INITIALIZATION_FAILED bug check has a value of 0x0000012F. This indicates that an initialization failure occurred while attempting to boot from a VHD.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

VHD_BOOT_INITIALIZATION_FAILED Parameters

PARAMETER	DESCRIPTION
1	Action that failed 1 : Couldn't extract VHD information from boot device. 2 : Timeout waiting for VHD parent device to surface. 3 : VHD path string memory allocation error. 4 : VHD path construction failed. 5 : VHD boot device mount failed. 6 : Disable sleep states failed. 7 : VHD information memory allocation error. 8 : VHD information construction failed.
2	NT status code
3	Reserved
4	Reserved

Bug Check 0x130: DYNAMIC_ADD_PROCESSOR_MISMATCH

7/2/2019 • 2 minutes to read • Edit Online

The DYNAMIC_ADD_PROCESSOR_MISMATCH bug check has a value of 0x00000130. This bugcheck indicates that a new processor added to the system is incompatible with the current configuration.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

DYNAMIC_ADD_PROCESSOR_MISMATCH Parameters

None

Bug Check 0x131: INVALID_EXTENDED_PROCESSOR_STATE

7/2/2019 • 2 minutes to read • Edit Online

The INVALID_EXTENDED_PROCESSOR_STATE bug check has a value of 0x00000131. This indicates that an invalid combination of parameters was detected while saving or restoring extended processor state.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

INVALID_EXTENDED_PROCESSOR_STATE Parameters

Parameter one indicates which validity check failed.

PARAMETER	DESCRIPTION
1	0 - Invalid feature mask was passed or extended processor state is not enabled.
2	Nonzero if extended state is enabled.
3	The low 32 bits of the feature mask.
4	The high 32 bits of the feature mask.

PARAMETER	DESCRIPTION
1	1 - An attempt to save or restore extended state was made at IRQL higher than DISPATCH_LEVEL.
2	The IRQL
3	Reserved
4	Reserved

PARAMETER	DESCRIPTION
1	2 - The previously saved state is for an equal or higher level.
2	The saved level.
3	The current level.
4	Reserved

PARAMETER	DESCRIPTION
1	3 - The previously saved state is for a different thread.
2	The saved thread.
3	The current thread.
4	Reserved

PARAMETER	DESCRIPTION
1	4 - Previously saved state is for a different level.
2	The saved level.
3	The current level.
4	Reserved

Bug Check 0x132: RESOURCE_OWNER_POINTER_INVALID

7/2/2019 • 2 minutes to read • Edit Online

The RESOURCE_OWNER_POINTER_INVALID bug check has a value of 0x00000132. This indicates that an invalid resource owner pointer was supplied.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

RESOURCE_OWNER_POINTER_INVALID Parameters

PARAMETER	DESCRIPTION
1	Resource
2	Resource->OwnerTable
3	CurrentThread
4	OwnerPointer

Bug Check 0x133 DPC_WATCHDOG_VIOLATION

7/2/2019 • 3 minutes to read • Edit Online

The DPC_WATCHDOG_VIOLATION bug check has a value of 0x00000133. This bug check indicates that the DPC watchdog executed, either because it detected a single long-running deferred procedure call (DPC), or because the system spent a prolonged time at an interrupt request level (IRQL) of DISPATCH_LEVEL or above. The value of Parameter 1 indicates whether a single DPC exceeded a timeout, or whether the system cumulatively spent an extended period of time at IRQL DISPATCH_LEVEL or above. DPCs should not run longer than 100 microseconds and ISRs should not run longer than 25 microseconds, however the actual timeout values on the system are set much higher.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

DPC_WATCHDOG_VIOLATION Parameters

Parameter 1 indicates the type of violation. The meaning of the other parameters depends on the value of *Parameter 1*.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0	The DPC time count (in ticks)	The DPC time allotment (in ticks).	Reserved	A single DPC or ISR exceeded its time allotment. The offending component can usually be identified with a stack trace.
1	The watchdog period	Reserved	Reserved	The system cumulatively spent an extended period of time at IRQL DISPATCH_LEVEL or above. The offending component can usually be identified with a stack trace.

Cause

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause.

Parameter 1 = 0

In this example, the tick count of 501 exceeds the DPC time allotment of 500. The image name indicates that this code was executing when the bug check occurred.

```
0: kd> !analyze -v
*******************************************************************************
*                                                                             *
*                        Bugcheck Analysis                                    *
*                                                                             *
*******************************************************************************

DPC_WATCHDOG_VIOLATION (133)
The DPC watchdog detected a prolonged run time at an IRQL of DISPATCH_LEVEL
or above.
Arguments:
Arg1: 0000000000000000, A single DPC or ISR exceeded its time allotment. The offending
    component can usually be identified with a stack trace.
Arg2: 0000000000000501, The DPC time count (in ticks).
Arg3: 0000000000000500, The DPC time allotment (in ticks).
Arg4: 0000000000000000

...

IMAGE_NAME:  BthA2DP.sys
...
```

Use the following debugger commands to gather more information for failures with a parameter of 0:

k (Display Stack Backtrace) to look at what code was running when the stop code occurred.

You may want to use the u, ub, uu (Unassemble) command to look deeper into the specifics of a the code that was running.

The **!pcr** extension displays the current status of the Processor Control Region (PCR) on a specific processor. In the output will be the address of the Prcb

```
                    Prcb: fffff80309974180
```

You can use the **dt (Display Type)** command to display additional information about the DPCs and the DPC Watchdog. For the address use the Prcb listed in the !pcr output:

```
dt nt!_KPRCB fffff80309974180 Dpc*
```

Parameter 1 = 1

For parameter of 1, the code may not stop in the offending area of code. In this case one approach is to use the event tracing to attempt to track down which driver is exceeding it's normal execution duration.

For more information see the following topics:

Crash dump analysis using the Windows debuggers (WinDbg)

Analyzing a Kernel-Mode Dump File with WinDbg

Using the !analyze Extension and !analyze

Remarks

In general this stop code is caused by faulty driver code that under certain conditions, does not complete its work within the allotted time frame.

If you are not equipped to use the Windows debugger to this problem, you should use some basic troubleshooting techniques.

- If a driver is identified in the bug check message, to isolate the issue, disable the driver. Check with the manufacturer for driver updates.
- Check the System Log in Event Viewer for additional error messages that might help identify the device or driver that is causing bug check 0x133.
- Confirm that any new hardware that is installed is compatible with the installed version of Windows. For example, you can get information about required hardware at Windows 10 Specifications.
- For additional general troubleshooting information, see **Blue Screen Data**.

Bug Check 0x134: DRIVE_EXTENDER

7/2/2019 • 2 minutes to read • Edit Online

The DRIVE_EXTENDER bug check has a value of 0x00000134. This indicates that the drive extender component has experienced a severe internal error that prevents continued system operation.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

DRIVE_EXTENDER Parameters

None

Bug Check 0x135: REGISTRY_FILTER_DRIVER_EXCEPTION

7/2/2019 • 2 minutes to read • Edit Online

The REGISTRY_FILTER_DRIVER_EXCEPTION bug check has a value of 0x00000135. This bugcheck is caused by an unhandled exception in a registry filtering driver.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

REGISTRY_FILTER_DRIVER_EXCEPTION Parameters

PARAMETER	DESCRIPTION
1	Exception Code
2	Address of the context record for the exception that caused the bugcheck
3	The driver's callback routine address
4	Reserved

Cause

This bugcheck indicates that a registry filtering driver didn't handle an exception inside its notification routine.

Resolution

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause. Identify the offending driver by using the 3rd parameter.

Bug Check 0x136: VHD_BOOT_HOST_VOLUME_NOT_ENOUGH_SPACE

7/2/2019 • 2 minutes to read • Edit Online

The VHD_BOOT_HOST_VOLUME_NOT_ENOUGH_SPACE bug check has a value of 0x00000136. This indicates that an initialization failure occurred while attempting to boot from a VHD. The volume that hosts the VHD does not have enough free space to expand the VHD.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

VHD_BOOT_HOST_VOLUME_NOT_ENOUGH_SPACE Parameters

PARAMETER	DESCRIPTION
1	0 : Unable to expand VHD file to full size.
2	NT Status Code
3	Reserved
4	Reserved

Bug Check 0x137: WIN32K_HANDLE_MANAGER

7/2/2019 • 2 minutes to read • Edit Online

The WIN32K_HANDLE_MANAGER bug check has a value of 0x00000137. This indicates that the win32k/ntuser handle manager has detected a fatal error.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

WIN32K_HANDLE_MANAGER Parameters

PARAMETER	DESCRIPTION
1	Reserved
2	Address of the object (If available)
3	Reserved
4	Reserved

Under some circumstances, additional parameters not listed here may be returned.

Bug Check 0x138: GPIO_CONTROLLER_DRIVER_ERROR

7/2/2019 • 2 minutes to read • Edit Online

The GPIO_CONTROLLER_DRIVER_ERROR bug check has a value of 0x00000138. This bug check indicates that the GPIO class extension driver encountered a fatal error.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

GPIO_CONTROLLER_DRIVER_ERROR Parameters

Parameter 1 indicates the type of violation. The meaning of the other parameters depends on the value of *Parameter 1*.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
1	GSIV	Reserved	Reserved	The GPIO controller managing the specific GSIV is not registered.
2	Context value	Reserved	Reserved	A client driver specified an invalid context to a lock or unlock request.
3	Indicates whether a critical transition is being requested.	Indicates whether the bank is already in F1 due to a non-critical transition.	Indicates whether the bank is already in F1 due to a critical transition.	PoFx requested that the GPIO controller send a bank through an inappropriate F1 power state and/or a critical transition.
4	Indicates whether a critical transition is being requested.	Indicates whether the bank is in F1 due to a non-critical transition.	Indicates whether the bank is in F1 due to a critical transition.	PoFx requested that the GPIO controller send a bank through an inappropriate F0 power state and/or a critical transition.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
5	NTSTATUS	GPIO device extension	GPIO interrupt parameters	An on-Soc GPIO interrupt operation failed.
6	NTSTATUS	GPIO device extension	GPIO IO parameters	An on-Soc GPIO IO operation failed.
7	Revision ID	Function Index	Reserved	A _DSM method returned malformed data.

Bug Check 0x139: KERNEL_SECURITY_CHECK_FAILURE

7/2/2019 • 6 minutes to read • Edit Online

The KERNEL_SECURITY_CHECK_FAILURE bug check has a value of 0x00000139. This bug check indicates that the kernel has detected the corruption of a critical data structure.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Bug Check 0x139 KERNEL_SECURITY_CHECK_FAILURE Parameters

PARAMETER	DESCRIPTION
1	The type of corruption. For more information, see the following table.
2	Address of the trap frame for the exception that caused the bug check
3	Address of the exception record for the exception that caused the bug check
4	Reserved

The following table describes possible values for Parameter 1.

PARAMETER 1	DESCRIPTION
0	A stack-based buffer has been overrun (legacy /GS violation).
1	VTGuard instrumentation code detected an attempt to use an illegal virtual function table. Typically, a C++ object was corrupted, and then a virtual method call was attempted using the corrupted object's **this** pointer.
2	Stack cookie instrumentation code detected a stack-based buffer overrun (/GS violation).
3	A LIST_ENTRY was corrupted (for example, a double remove). For more information, see the following Cause section.
4	Reserved
5	An invalid parameter was passed to a function that considers invalid parameters fatal.

PARAMETER 1	DESCRIPTION
6	The stack cookie security cookie was not properly initialized by the loader. This may be caused by building a driver to run only on Windows 8 and attempting to load the driver image on an earlier version of Windows. To avoid this problem, you must build the driver to run on an earlier version of Windows.
7	A fatal program exit was requested.
8	A array bounds check inserted by the compiler detected an illegal array indexing operation.
9	A call to **RtlQueryRegistryValues** was made specifying RTL_QUERY_REGISTRY_DIRECT without RTL_QUERY_REGISTRY_TYPECHECK, and the target value was not in a trusted system hive.
10	Indirect call guard check detected invalid control transfer.
11	Write guard check detected invalid memory write.
12	An attempt was made to switch to an invalid fiber context.
13	An attempt was made to assign an invalid register context.
14	The reference count for an object is invalid.
18	An attempt was made to switch to an invalid jmp_buf context.
19	An unsafe modification was made to read-only data.
20	A cryptographic self-test failed.
21	An invalid exception chain was detected.
22	A cryptographic library error occurred.
23	An invalid call was made from within DllMain.
24	An invalid image base address was detected.
25	An unrecoverable failure was encountered while protecting a delay load import.
26	A call was made to an unsafe extension.
27	A deprecated service was invoked.
28	An out of bounds buffer access was detected.
29	An RTL_BALANCED_NODE RBTree entry has been corrupted.
37	An out of range switch jumptable entry was invoked.

PARAMETER 1	DESCRIPTION
38	A longjmp was attempted to an invalid target.
39	An export suppressed call target couldn't be made a valid call target.

Cause

Using the parameter 1 table, and a dump file, it is possible to narrow down the cause for many bug checks of this type.

LIST_ENTRY corruption can be difficult to track down and this bug check, indicates that an inconsistency has been introduced into a doubly-linked list (detected when an individual list entry element is added to or removed from the list). Unfortunately, the inconsistency is not necessarily detected at the time when the corruption occurred, so some detective work may be necessary to identify the root cause.

Common causes of list entry corruption include:

- A driver has corrupted a kernel synchronization object, such as a KEVENT (for example double initializing a KEVENT while a thread was still waiting on that same KEVENT, or allowing a stack-based KEVENT to go out of scope while another thread was using that KEVENT). This type of bug check typically occurs in nt!Ke* or nt!Ki* code. It can happen when a thread finishes waiting on a synchronization object or when code attempts to put a synchronization object in the signaled state. Usually, the synchronization object being signaled is the one that has been corrupted. Sometimes, Driver Verifier with special pool can help track down the culprit (if the corrupted synchronization object is in a pool block that has already been freed).
- A driver has corrupted a periodic KTIMER. This type of bug check typically occurs in nt!Ke* or nt!Ki* code and involves signaling a timer, or inserting or removing a timer from a timer table. The timer being manipulated may be the corrupted one, but it might be necessary to inspect the timer table with **!timer** (or manually walking the timer list links) to identify which timer has been corrupted. Sometimes, Driver Verifier with special pool can help track down the culprit (if the corrupted KTIMER is in a pool block that has already been freed).
- A driver has mismanaged an internal LIST_ENTRY-style linked list. A typical example would be calling **RemoveEntryList** twice on the same list entry without reinserting the list entry between the two **RemoveEntryList** calls. Other variations are possible, such as double inserting an entry into the same list.
- A driver has freed a data structure that contains a LIST_ENTRY without removing the data structure from its corresponding list, causing corruption to be detected later when the list is examined after the old pool block has been reused.
- A driver has used a LIST_ENTRY-style list in a concurrent fashion without proper synchronization, resulting in a torn update to the list.

In most cases, you can identify the corrupted data structure by walking the linked list both forward and backwards (the **dl** and **dlb** commands are useful for this purpose) and comparing the results. Where the list is inconsistent between a forward and backward walk is typically the location of the corruption. Since a linked list update operation can modify the list links of a neighboring element, you should look at the neighbors of a corrupted list entry closely, as they may be the underlying culprit.

Because many system components internally utilize LIST_ENTRY lists, various types of resource mismanagement by a driver using system APIs might cause linked list corruption in a system-managed linked list.

Resolution

Determining the cause of this issues typically requires the use of the debugger to gather additional information. Multiple dump files should be examined to see if this stop code has similar characteristics, such as the code that is

running when the stop code appears.

For more information, see Crash dump analysis using the Windows debuggers (WinDbg), Using the !analyze Extension and !analyze.

Use the event log to see if there are higher level events that occur leading up to this stop code.

These general troubleshooting tips may be helpful.

- If you recently added hardware to the system, try removing or replacing it. Or check with the manufacturer to see if any patches are available.
- If new device drivers or system services have been added recently, try removing or updating them. Try to determine what changed in the system that caused the new bug check code to appear.
- Check the System Log in Event Viewer for additional error messages that might help pinpoint the device or driver that is causing the error. For more information, see Open Event Viewer. Look for critical errors in the system log that occurred in the same time window as the blue screen.
- Look in **Device Manager** to see if any devices are marked with the exclamation point (!). Review the events log displayed in driver properties for any faulting driver. Try updating the related driver.
- Run a virus detection program. Viruses can infect all types of hard disks formatted for Windows, and resulting disk corruption can generate system bug check codes. Make sure the virus detection program checks the Master Boot Record for infections.
- For additional general troubleshooting information, see **Blue Screen Data**.

See also

Crash dump analysis using the Windows debuggers (WinDbg)

Analyzing a Kernel-Mode Dump File with WinDbg

Bug Check 0x13A: KERNEL_MODE_HEAP_CORRUPTION

7/2/2019 • 2 minutes to read • Edit Online

The KERNEL_MODE_HEAP_CORRUPTION bug check has a value of 0x0000013A. This indicates that the kernel mode heap manager has detected corruption in a heap.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

KERNEL_MODE_HEAP_CORRUPTION Parameters

PARAMETER	DESCRIPTION
1	Type of corruption detected 0x3 : A corrupt entry header was detected. 0x4 : Multiple corrupt entry headers were detected. 0x5 : A corrupt entry header in a large allocation was detected. 0x6 : A corruption was detected with features consistent with a buffer overrun. 0x7 : A corruption was detected with features consistent with a buffer underrun. 0x8 : A free block was passed to an operation that is only valid for busy blocks. 0x9 : An invalid argument was specified for the current operation. 0xA : A corruption was detected with features consistent with a use-after-free error. 0xB : The wrong heap was specified for the current operation. 0xC : A corrupt free list was detected. 0xD : The heap detected list corruption in a list other than the free list.
2	Address of the heap that reported the corruption
3	Address at which the corruption was detected
4	Reserved

Bug Check 0x13B: PASSIVE_INTERRUPT_ERROR

7/2/2019 • 2 minutes to read • Edit Online

The PASSIVE_INTERRUPT_ERROR bug check has a value of 0x0000013B. This indicates that the kernel has detected issues with the passive-level interrupt.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

PASSIVE_INTERRUPT_ERROR Parameters

PARAMETER	DESCRIPTION
1	Type of error detected 0x1 : A driver tried to acquire an interrupt spinlock but passed in a passive-level interrupt object.
2	Address of the KINTERRUPT object for the passive-level interrupt.
3	Reserved
4	Reserved

Bug Check 0x13C: INVALID_IO_BOOST_STATE

7/2/2019 • 2 minutes to read • Edit Online

The INVALID_IO_BOOST_STATE bug check has a value of 0x0000013C. This indicates that a thread exited with an invalid I/O boost state. This should be zero when a thread exits.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

INVALID_IO_BOOST_STATE Parameters

PARAMETER	DESCRIPTION
1	Pointer to the thread which had the invalid boost state
2	Current boost state or throttle count
3	Reserved
4	Reserved

Bug Check 0x13D: CRITICAL_INITIALIZATION_FAILURE

7/2/2019 • 2 minutes to read • Edit Online

The CRITICAL_INITIALIZATION_FAILURE bug check has a value of 0x0000013D. This indicates that early kernel initialization has failed.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

CRITICAL_INITIALIZATION_FAILURE Parameters

PARAMETER	DESCRIPTION
1	Reserved
2	Reserved
3	Reserved
4	Reserved

Bug Check 0x140: STORAGE_DEVICE_ABNORMALITY_DETECTED

7/2/2019 • 2 minutes to read • Edit Online

The STORAGE_DEVICE_ABNORMALITY_DETECTED bug check has a value of 0x00000140. This indicates that the storage driver stack encountered rate of responsiveness violations, exceeding the threshold, or other failures to respond.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

STORAGE_DEVICE_ABNORMALITY_DETECTED Parameters

None

Bug Check 0x141: VIDEO_ENGINE_TIMEOUT_DETECTED

10/8/2019 • 2 minutes to read • Edit Online

The VIDEO_ENGINE_TIMEOUT_DETECTED bug check has a value of 0x00000141. This indicates that one of the display engines failed to respond in timely fashion.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

VIDEO_ENGINE_TIMEOUT_DETECTED Parameters

PARAMETER	DESCRIPTION
1	Optional pointer to internal TDR recovery context (TDR_RECOVERY_CONTEXT).
2	The pointer into responsible device driver module (e.g owner tag).
3	The secondary driver specific bucketing key.
4	Optional internal context dependent data.

Remarks

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause. Secondary data of tag {270A33FD-3DA6-460D-BA89-3C1BAE21E39B} contains additional TDR related data. Use **.enumtag** to view the data.

Bug Check 0x142: VIDEO_TDR_APPLICATION_BLOCKED

7/13/2019 • 2 minutes to read • Edit Online

The VIDEO_TDR_APPLICATION_BLOCKED bug check has a value of 0x00000142. This indicates that an application has been blocked from accessing graphics hardware.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

VIDEO_TDR_APPLICATION_BLOCKED Parameters

PARAMETER	DESCRIPTION
1	Optional pointer to internal TDR recovery context (TDR_RECOVERY_CONTEXT).
2	The pointer into responsible device driver module (e.g owner tag).
3	The secondary driver specific bucketing key.
4	Id of the process being blocked from accessing the GPU.

Remarks

The **!analyze** debug extension displays information about the bug check and can be very helpful in determining the root cause. Secondary data of tag {270A33FD-3DA6-460D-BA89-3C1BAE21E39B} contains additional TDR related data. Use **.enumtag (Enumerate Secondary Callback Data)** to view the data.

Bug Check 0x143: PROCESSOR_DRIVER_INTERNAL

7/2/2019 • 2 minutes to read • Edit Online

The PROCESSOR_DRIVER_INTERNAL bug check has a value of 0x00000143. This indicates that the Processor Power Management (PPM) driver encountered a fatal error.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

PROCESSOR_DRIVER_INTERNAL Parameters

PARAMETER	DESCRIPTION
1	1 - Power Engine Plugin(PEP) failed to accept a required notification
2	PEP runtime Notification type
3	Pointer to notification message
4	Pointer to processor device context (FDO_DATA) issuing the notification

PARAMETER	DESCRIPTION
1	2 - Power Engine Plugin (PEP) returned invalid processor idle state
2	Type of invalid state 0x0 : PEP requested too many processors for coordinated idle state Parameter 3 - Number of processors requested to participate in coordinated idle transitions Parameter 4 - Pointer to processor device context (FDO_DATA) 0x1 : PEP requested processor to be in an invalid idle state Parameter 3 - Idle state index requested Parameter 4 - Pointer to processor device context (FDO_DATA) corresponding to the invalid idle state 0x2 : PEP requested the platform to be in an invalid idle state Parameter 3 - Platform idle state index requested Parameter 4 - Pointer to processor device context (FDO_DATA) corresponding to the invalid idle state
3	Refer to parameter 2

PARAMETER	DESCRIPTION
4	Refer to parameter 2

Cause

The processor driver detected an irreconcilable condition which prompted it to bugcheck. This likely happens during the processor idle and perf-state change execution, which may involve other entities such has kernel, HAL and the Power Engine Plugin (PEP). Information from bugcheck will help identify which of the assumptions made by the processor driver in dealing with other entities was violated. The root cause may lie in other entities and a dump file may reveal more information to ascertain the reason for the bugcheck.

Bug Check 0x144: BUGCODE_USB3_DRIVER

10/23/2019 • 6 minutes to read • Edit Online

The **BUGCODE_USB3_DRIVER** bug check has a value of 0x00000144. This is the code used for all USB 3 bug checks. Parameter 1 specifies the type of the USB 3 bug check, and the meanings of the other parameters are dependent on Parameter 1.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

BUGCODE_USB3_DRIVER Parameters

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x1	Optional. Pointer to the IRP used to resend the URB	Pointer to the URB	Pointer to the client driver's device object	A client driver used an URB that it had previously sent to the core stack.
0x2	Pointer to the physical device object (PDO) for the boot device	Reserved	Reserved	A boot or paging device failed re-enumeration.
0x3	Optional. Pointer to the IRP used to send the URB	Pointer to the corrupted URB	Pointer to the client driver's device object	A client driver sent a corrupted URB to the core stack. This can happen because the client driver did not allocate the URB using **USBD_*xxx*UrbAllocate** or because the client driver did a buffer underrun for the URB.
0x800	IRQL at which the Open Static Streams request was sent	Pointer to the Open Static Streams IRP	Pointer to the client driver's device object	An Open Static Streams request was sent at IRQL > PASSIVE LEVEL.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x801	Pointer to the Open Static Streams IRP	Pointer to the Open Static Streams URB	Pointer to the client driver's device object	A client driver attempted to open static streams before querying for streams capability. A client driver cannot open a static stream until after it successfully queries for the streams capability. For more information, see Remarks.
0x802	Number of static streams that the client driver tried to open	Number of static streams that were granted to the client driver	Pointer to the client driver's device object	A Client driver tried to open an invalid number of static streams. The number of streams cannot be 0 and cannot be greater than the value returned to the client driver in the query USB capability call.
0x803	Pointer to the Open Static Streams IRP	Pointer to the Open Static Streams URB	Pointer to the client driver's device object	A client driver attempted to open static streams for an endpoint that already had static streams open. Before opening static streams, the client driver must close the previously opened static streams.
0x804	The leaked handle context. Run **!usbanalyze -v** to get information about the leaked handle and URBs. You must enable Driver Verifier for the client driver.	Device object passed to **USBD_CreateHandle**.	Reserved	A client driver forgot to close a handle it created earlier using **USBD_CreateHandle** or forgot to free an URB it allocated.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x805	WDFREQUEST handle for the Close Static Streams URB	Pointer to the Close Static Streams URB	Pointer to the client driver's device object	A client driver sent a Close Static Streams URB in an invalid state (for example, after processing D0 Exit).
0x806	Pointer to the IRP	Pointer to the URB	Pointer to the client driver's device object	A client driver attempted to send a chained **MDL** before querying for chained **MDL** capability. A client driver cannot send a chained **MDL** until after it successfully queries for the chained **MDL** capability. For more information, see Remarks.
0x807	Pointer to the chained **MDL**	Pointer to the URB	Pointer to the client driver's device object if available	A client driver sent an URB to the core stack with a transfer buffer length longer than the byte count (returned by **MmGetMdlByteCount**) of the **MDL** passed in. For more information, see Remarks.
0x1001	XHCI_LIVEDUMP_CONTEXT	Reserved	Reserved	The xHCI controller asserted the HSE bit, which indicates a host system error.
0x1002	XHCI_LIVEDUMP_CONTEXT	Reserved	Reserved	The xHCI controller asserted the HCE bit, which indicates a host controller error.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x1003	XHCI_LIVEDUMP_CONTEXT	Reserved	Reserved	The xHCI stop endpoint command returned an unhandled completion code.
0x1004	XHCI_LIVEDUMP_CONTEXT	Reserved	Reserved	The xHCI endpoint state received a context state error after an xHCI endpoint stop command was issued.
0x1005	XHCI_LIVEDUMP_CONTEXT	Reserved	Reserved	Set dequeue pointer failed during an attempt to clear stall on control endpoint.
0x1006	XHCI_LIVEDUMP_CONTEXT	Reserved	Reserved	Reset EP failed during an attempt to clear stall on control endpoint.
0x1007	XHCI_LIVEDUMP_CONTEXT	Reserved	Reserved	The reset of the xHCI controller failed during reset recovery.
0x1008	XHCI_LIVEDUMP_CONTEXT	Reserved	Reserved	The restart of the xHCI controller failed during reset recovery.
0x1009	XHCI_LIVEDUMP_CONTEXT	Reserved	Reserved	An xHCI controller command failed to complete after the command timeout abort.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x100A	XHCI_LIVEDUMP_CONTEXT	Reserved	Reserved	Set dequeue pointer failed during an attempt to set the dequeue pointer after endpoint stop completion.
0x100B	XHCI_LIVEDUMP_CONTEXT	Reserved	Reserved	The stop of the xHCI controller failed during reset recovery.
0x100C	XHCI_LIVEDUMP_CONTEXT	Reserved	Reserved	The firmware in the xHCI controller is not supported. The xHCI driver will not load on this controller unless the firmware is updated.
0x100D	XHCI_LIVEDUMP_CONTEXT	Reserved	Reserved	The controller was detected to be physically removed.
0x100E	XHCI_LIVEDUMP_CONTEXT	Reserved	Reserved	The driver detect an error on a stream enabled endpoint.
0x100F	XHCI_LIVEDUMP_CONTEXT	Reserved	Reserved	The firmware in the xHCI controller is outdated. The xHCI driver will continue working with this controller but may run into some issues. A firmware update is recommended.
0x1010	XHCI_LIVEDUMP_CONTEXT	Reserved	Reserved	A transfer event TRB completed with an unhandled completion code.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x1011	XHCI_LIVEDUMP_CONTEXT	Reserved	Reserved	The controller reported that the event ring became full. The controller is also known to drop events when this happens.
0x1012	XHCI_LIVEDUMP_CONTEXT	Reserved	Reserved	The controller completed a command out of order.
0x1013	XHCI_LIVEDUMP_CONTEXT	Reserved	Reserved	After command abort completion, the command ring dequeue pointer reported by the controller is incorrect.
0x1014	XHCI_LIVEDUMP_CONTEXT	Reserved	Reserved	After enable slot completion, controller gave us a bad slot id.
0x1015	XHCI_LIVEDUMP_CONTEXT	Reserved	Reserved	Controller failed a SetAddress command with BSR1. That is unexpected.
0x1016	XHCI_LIVEDUMP_CONTEXT	Reserved	Reserved	Controller failed to enable a slot during a usbdevice reset. This is unexpected.
0x1017	XHCI_LIVEDUMP_CONTEXT	Reserved	Reserved	Controller failed an endpoints configure command where we were deconfiguring the endpoints. That is unexpected.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x1018	XHCI_LIVEDUMP_CONTEXT	Reserved	Reserved	Controller failed a disable slot command. That is unexpected.
0x1019	XHCI_LIVEDUMP_CONTEXT	Reserved	Reserved	Controller failed a USB device reset command. That is unexpected.
0x101A	XHCI_LIVEDUMP_CONTEXT	Reserved	Reserved	After endpoint reset, Set Dequeue Pointer command failed.
0x101B	XHCI_LIVEDUMP_CONTEXT	Reserved	Reserved	The xHCI reset endpoint command returned an unhandled completion code.
0x101C	XHCI_LIVEDUMP_CONTEXT	Reserved	Reserved	The D0Entry for xHCI failed.
0x101D	XHCI_LIVEDUMP_CONTEXT	Reserved	Reserved	Temporarily dropping and adding a stream endpoint (as two commands) failed, when using the Configure Endpoint command instead of Set Dequeue Pointer during request cancellation.
0x101E	XHCI_LIVEDUMP_CONTEXT	Reserved	Reserved	The controller indicated a transfer completion that was not pending on the controller. EventData == 1 (dereferencing the Transfer Event TRB's pointer would have caused a bugcheck)

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x101F	XHCI_LIVEDUMP_CONTEXT	Reserved	Reserved	The controller indicated a transfer completion that was not pending on the controller. EventData == 0 (logical address in transfer event TRB not matched)
0x1020	XHCI_LIVEDUMP_CONTEXT	Reserved	Reserved	The controller indicated a transfer completion that was not pending on the controller. EventData == 0 (logical address in transfer event TRB not matched) The Transfer Event TRB may be redundant (points somewhere near a recently completed request).
0x1021	XHCI_LIVEDUMP_CONTEXT	Reserved	Reserved	Temporarily dropping and adding a stream endpoint (as two commands) failed, when using the Configure Endpoint command as part of resetting an endpoint that was not Halted.
0x1022	XHCI_LIVEDUMP_CONTEXT	Reserved	Reserved	Dropping and adding the same endpoint (as one command) failed.
0x3000	USBHUB3_LIVEDUMP_CONTEXT	Reserved	Reserved	A misbehaving hub was successfully reset by the hub driver.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4	CAUSE OF ERROR
0x3001	USBHUB3_LIVED UMP_CONTEXT	Reserved	Reserved	A misbehaving hub failed to be reset successfully by the hub driver.
0x3002	USBHUB3_LIVED UMP_CONTEXT	Reserved	Reserved	A non-function SuperSpeed hub was disabled by the hub driver.
0x3003	USBHUB3_LIVED UMP_CONTEXT	Reserved	Reserved	A USB device failed enumeration.

Remarks

To query for a USB capability, the client driver must call **WdfUsbTargetDeviceQueryUsbCapability** or **USBD_QueryUsbCapability**

To send a chained **MDL**, the client driver must call **USBD_QueryUsbCapability** and use **URB_FUNCTION_BULK_OR_INTERRUPT_TRANSFER_USING_CHAINED_MDL** or **URB_FUNCTION_ISOCH_TRANSFER_USING_CHAINED_MDL**.

See also

Universal Serial Bus (USB)

Bug Check 0x145: SECURE_BOOT_VIOLATION

7/2/2019 • 2 minutes to read • Edit Online

The SECURE_BOOT_VIOLATION bug check has a value of 0x00000145. This indicates that the secure Boot policy enforcement could not be started due to an invalid policy or a required operation not being completed.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

SECURE_BOOT_VIOLATION Parameters

PARAMETER	DESCRIPTION
1	The status code of the failure.
2	Address of the Secure Boot policy.
3	Size of the Secure Boot policy.
4	Reserved

Bug Check 0x147: ABNORMAL_RESET_DETECTED

7/2/2019 • 2 minutes to read • Edit Online

The ABNORMAL_RESET_DETECTED bug check has a value of 0x00000147. This indicates that Windows underwent an abnormal reset. No context or exception records were saved, and bugcheck callbacks were not called.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

ABNORMAL_RESET_DETECTED Parameters

None

Bug Check 0x14B: SOC_SUBSYSTEM_FAILURE

10/23/2019 • 2 minutes to read • Edit Online

The SOC_SUBSYSTEM_FAILURE bug check has a value of 0x0000014B. This indicates that an unrecoverable error was encountered in a System on a Chip (SoC) subsystem.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Bug Check 0x14B SOC_SUBSYSTEM_FAILURE Parameters

PARAMETER	DESCRIPTION
1	Address of an **SOC_SUBSYSTEM_FAILURE_DETAILS** structure.
2	Reserved.
3	Reserved.
4	Optional. Address of a vendor-supplied data block.

Resolution

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause.

```
2: kd> !analyze -v
*******************************************************************************
*                                                                             *
*                        Bugcheck Analysis                                    *
*                                                                             *
*******************************************************************************

SOC_SUBSYSTEM_FAILURE (14b)
A SOC subsystem has experienced an unrecoverable critical fault.
Arguments:
Arg1: 9aa8d630, nt!SOC_SUBSYSTEM_FAILURE_DETAILS
Arg2: 00000000, Reserved
Arg3: 00000000, Reserved
Arg4: a126c000, (Optional) address to vendor supplied general purpose data block.
```

Use the provided nt!SOC_SUBSYSTEM_FAILURE_DETAILS structure to dump the failure data using the dt command and the address provided by Arg1.

```
2: kd> dt nt!SOC_SUBSYSTEM_FAILURE_DETAILS 9aa8d630
   +0x000 SubsysType       : 1 ( SOC_SUBSYS_AUDIO_DSP )
   +0x008 FirmwareVersion  : 0
   +0x010 HardwareVersion  : 0
   +0x018 UnifiedFailureRegionSize : 0x24
   +0x01c UnifiedFailureRegion : [1]  "F"
```

Work with SoC vendor to further parse the data, including the optional vendor supplied general purpose data block.

You may want to examine the stack trace using the **k, kb, kc, kd, kp, kP, kv (Display Stack Backtrace)** command. You can specify the processor number to examine the stacks on all processors.

You can also set a breakpoint in the code leading up to this stop code and attempt to single step forward into the faulting code.

For more information see the following topics:

Crash dump analysis using the Windows debuggers (WinDbg)

If you are not equipped to use the Windows debugger to work on this problem, you can use some basic troubleshooting techniques.

- Check the System Log in Event Viewer for additional error messages that might help identify the device or driver that is causing this bug check.
- If a driver is identified in the bug check message, disable the driver or check with the manufacturer for driver updates.
- You can try running the hardware diagnostics supplied by the system manufacturer.
- For additional general troubleshooting information, see **Blue Screen Data**.

Requirements

Minimum supported client	Windows 8
Minimum supported server	Windows Server 2012

Bug Check 0x149: REFS_FILE_SYSTEM

7/2/2019 • 2 minutes to read • Edit Online

The REFS_FILE_SYSTEM bug check has a value of 0x00000149. This indicates that a file system error has occurred.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

REFS_FILE_SYSTEM Parameters

PARAMETER	DESCRIPTION
1	__LINE__
2	ExceptionRecord
3	ContextRecord
4	ExceptionRecord->ExceptionAddress

PARAMETER	DESCRIPTION
1	Message
2	Reserved
3	Reserved
4	Reserved

Resolution

If you see RefsExceptionFilter on the stack then the 2nd and 3rd parameters are the exception record and context record. Do a **.exr** on the 2nd parameter to view the exception information, then do a **.cxr** on the 3rd parameter and **kb** to obtain a more informative stack trace.

Bug Check 0x14A: KERNEL_WMI_INTERNAL

7/2/2019 • 2 minutes to read • Edit Online

The KERNEL_WMI_INTERNAL bug check has a value of 0x0000014A. This indicates that the internal kernel WMI subsystem has encountered a fatal error.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

KERNEL_WMI_INTERNAL Parameters

PARAMETER	DESCRIPTION
1	0 : A kernel WMI entry reference count was incremented from 0. Parameter 2: Pointer to the kernel WMI entry. 1 : A kernel WMI datasource was removed prematurely. Parameter 2: Pointer to the kernel WMI datasource.
2	See parameter 1
3	Reserved
4	Reserved

Bug Check 0x14C: FATAL_ABNORMAL_RESET_ERROR

7/2/2019 • 2 minutes to read • Edit Online

The FATAL_ABNORMAL_RESET_ERROR bug check has a value of 0x0000014C. This indicates that an unrecoverable system error occurred or the system has abnormally reset.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

FATAL_ABNORMAL_RESET_ERROR Parameters

None

Cause

The system encountered an unexpected error and restarted. Issues that may cause this error include: hardware watchdog timer in application or auxiliary processors indicating a system hang, user-initiated key sequence because of a hang, a brownout, or failures in the default bugcheck path. The cache may not be flushed and the resulting full memory dump may not contain the current thread context.

Secondary data of tag {E1D08891-D5A3-45F9-B811-AD711DFB2607} contains additional "Blackbox" data. Use **.enumtag (Enumerate Secondary Callback Data)** to view the data.

Bug Check 0x14D: EXCEPTION_SCOPE_INVALID

7/2/2019 • 2 minutes to read • Edit Online

The EXCEPTION_SCOPE_INVALID bug check has a value of 0x0000014D. This indicates that an internal inconsistency in exception dispatching has been detected.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

EXCEPTION_SCOPE_INVALID Parameters

PARAMETER	DESCRIPTION
1	Reserved
2	Reserved
3	Reserved
4	Reserved

Bug Check 0x14E: SOC_CRITICAL_DEVICE_REMOVED

7/2/2019 • 2 minutes to read • Edit Online

The SOC_CRITICAL_DEVICE_REMOVED bug check has a value of 0x0000014E. This indicates that a critical SOC device has been unexpectedly removed or failed.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

SOC_CRITICAL_DEVICE_REMOVED Parameters

PARAMETER	DESCRIPTION
1	When available, indicates the ID of the device which was removed (4 character packed code)
2	Reserved
3	Reserved
4	Reserved

Bug Check 0x14F: PDC_WATCHDOG_TIMEOUT

7/2/2019 • 2 minutes to read • Edit Online

The PDC_WATCHDOG_TIMEOUT bug check has a value of 0x0000014F. This indicates that a system component failed to respond within the allocated time period, preventing the system from exiting connected standby.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

PDC_WATCHDOG_TIMEOUT Parameters

PARAMETER	DESCRIPTION
1	Client ID of the hung component.
2	Client type of the hung component. 0x1 : A notification client failed to respond. Parameter 3: Pointer to the notification client (PDC_NOTIFICATION_CLIENT). Parameter 4: Pointer to a pdc!PDC_14F_TRIAGE structure. 0x2 : A resiliency client failed to respond. Parameter 3: Pointer to the resiliency client (PDC_RESILIENCY_CLIENT). Parameter 4: Pointer to a pdc!PDC_14F_TRIAGE structure. 0x3 : An activator client held a reference for too long. Parameter 3: Pointer to the activation client (pdc!_PDC_ACTIVATOR_CLIENT). Parameter 4: Pointer to a pdc!PDC_14F_TRIAGE structure. 0x100 : Win32k did not complete a monitor-on request in a timely manner. Parameter 3: The most recent POWER_MONITOR_REQUEST_REASON value for this request. Parameter 4: A value indicating the internal path taken to initiate the request.
3	See parameter 2
4	See parameter 2

Resolution

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause.

Bug Check 0x150: TCPIP_AOAC_NIC_ACTIVE_REFERENCE_LEAK

7/2/2019 • 2 minutes to read • Edit Online

The TCPIP_AOAC_NIC_ACTIVE_REFERENCE_LEAK bug check has a value of 0x00000150. This indicates that the NIC active reference should have been released when the send queue was fully drained.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

TCPIP_AOAC_NIC_ACTIVE_REFERENCE_LEAK Parameters

None

Bug Check 0x151: UNSUPPORTED_INSTRUCTION_MODE

7/2/2019 • 2 minutes to read • Edit Online

The UNSUPPORTED_INSTRUCTION_MODE bug check has a value of 0x00000151. This indicates that an attempt was made to execute code using an unsupported processor instruction mode (for example, executing classic ARM instructions instead of ThumbV2 instructions). This is not permitted.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

UNSUPPORTED_INSTRUCTION_MODE Parameters

PARAMETER	DESCRIPTION
1	Program counter when the problem was detected.
2	Trap Frame
3	Reserved
4	Reserved

Bug Check 0x152: INVALID_PUSH_LOCK_FLAGS

7/2/2019 • 2 minutes to read • Edit Online

The INVALID_PUSH_LOCK_FLAGS bug check has a value of 0x00000152. This indicates that the flags supplied to one of push lock APIs were invalid.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

INVALID_PUSH_LOCK_FLAGS Parameters

PARAMETER	DESCRIPTION
1	The invalid flags supplied by the caller
2	The address of the push lock
3	Reserved
4	Reserved

Bug Check 0x153: KERNEL_LOCK_ENTRY_LEAKED_ON_THREAD_TERMINATION

7/2/2019 • 2 minutes to read • Edit Online

The KERNEL_LOCK_ENTRY_LEAKED_ON_THREAD_TERMINATION bug check has a value of 0x00000153. This indicates that a thread was terminated before it had freed all its AutoBoost lock entries.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

KERNEL_LOCK_ENTRY_LEAKED_ON_THREAD_TERMINATION Parameters

PARAMETER	DESCRIPTION
1	The address of the thread
2	The address of the entry that was not freed
3	A status code indicating the state of the entry 0x1 : Lock pointer was not NULL 0x2 : Thread pointer reserved bits were set 0x3 : Thread pointer was corrupted 0x4 : The entry had residual IO or CPU boosts left
4	Reserved

Cause

This is typically caused when a thread never released a lock it previously acquired (e.g. by relying on another thread to release it), or if the thread did not supply a consistent set of flags to lock package APIs.

Bug Check 0x154: UNEXPECTED_STORE_EXCEPTION

7/2/2019 • 2 minutes to read • Edit Online

The UNEXPECTED_STORE_EXCEPTION bug check has a value of 0x00000154. This indicates that the store component caught an unexpected exception.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

UNEXPECTED_STORE_EXCEPTION Parameters

PARAMETER	DESCRIPTION
1	Pointer to the store context or data manager
2	Exception information
3	Reserved
4	Reserved

Bug Check 0x155: OS_DATA_TAMPERING

7/2/2019 • 2 minutes to read • Edit Online

The OS_DATA_TAMPERING bug check has a value of 0x00000155.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

OS_DATA_TAMPERING Parameters

None

Bug Check 0x156: WINSOCK_DETECTED_HUNG_CLOSESOCKET_LIVEDUMP

7/2/2019 • 2 minutes to read • Edit Online

The WINSOCK_DETECTED_HUNG_CLOSESOCKET_LIVEDUMP bug check has a value of 0x00000156. This indicates that Winsock detected a hung transport endpoint close request.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

WINSOCK_DETECTED_HUNG_CLOSESOCKET_LIVEDUMP Parameters

PARAMETER	DESCRIPTION
1	AFD endpoint pointer (!afdkd.endp <ptr>)
2	Transport endpoint type 0x1 : UDP datagram 0x2 : RAW datagram 0x3 : TCP listener 0x4 : TCP endpoint
3	Number of buffered send bytes for datagram endpoints
4	afd!NETIO_SUPER_TRIAGE_BLOCK

Cause

While processing a closesocket request, Winsock detected a hung transport endpoint close request. The system generated a live dump for analysis, then the closesocket request was completed without waiting for the completion of hung transport endpoint close request.

Bug Check 0x157: KERNEL_THREAD_PRIORITY_FLOOR_VIOLATION

7/2/2019 • 2 minutes to read • Edit Online

The ATTEMPTED_SWITCH_FROM_DPC bug check has a value of 0x00000157. This indicates that an illegal operation was attempted on the priority floor of a particular thread.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

KERNEL_THREAD_PRIORITY_FLOOR_VIOLATION Parameters

PARAMETER	DESCRIPTION
1	The address of the thread
2	The target priority value
3	A status code indicating the nature of the violation 0x1 : The priority counter for the target priority over-flowed 0x2 : The priority counter for the target priority under-flowed 0x3 : The target priority value was illegal
4	Reserved

Bug Check 0x158: ILLEGAL_IOMMU_PAGE_FAULT

7/2/2019 • 2 minutes to read • Edit Online

The ILLEGAL_IOMMU_PAGE_FAULT bug check has a value of 0x00000158. This indicates that the IOMMU has delivered a page fault packet for an invalid ASID. This is not safe since the ASID may have already been reused.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

ILLEGAL_IOMMU_PAGE_FAULT Parameters

PARAMETER	DESCRIPTION
1	The invalid ASID.
2	The number of ASIDs currently in use.
3	The process using this ASID.
4	The ASID's reference count.

Bug Check 0x159: HAL_ILLEGAL_IOMMU_PAGE_FAULT

7/2/2019 • 2 minutes to read • Edit Online

The HAL_ILLEGAL_IOMMU_PAGE_FAULT bug check has a value of 0x00000159. This indicates that the IOMMU has delivered a page fault against an ASID that was in the process of being freed. The driver was responsible for completing any inflight requests before this point in time and this bugcheck indicates a driver in the system did not do so.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

HAL_ILLEGAL_IOMMU_PAGE_FAULT Parameters

PARAMETER	DESCRIPTION
1	IOMMU Vendor disambiguation
2	Pointer to fault packet
3	Vendor specific fault packet data
4	Vendor specific fault packet data

PARAMETER	DESCRIPTION
1	IOMMU Vendor disambiguation = 0x3xxx.
2	Status
3	PASID
4	DirectoryBase

Bug Check 0x15A: SDBUS_INTERNAL_ERROR

7/2/2019 • 2 minutes to read • Edit Online

The SDBUS_INTERNAL_ERROR bug check has a value of 0x0000015A. This indicates that an unrecoverable hardware failure has occurred on an SD-attached device.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

SDBUS_INTERNAL_ERROR Parameters

PARAMETER	DESCRIPTION
1	Pointer to the internal SD work packet that caused the failure
2	Pointer the controller socket information
3	Pointer to the SD request packet sent down to the bus driver
4	Reserved

Bug Check 0x15B: WORKER_THREAD_RETURNED_WITH_SYSTEM_PAGE_PRIORITY_ACTIVE

7/2/2019 • 2 minutes to read • Edit Online

The WORKER_THREAD_RETURNED_WITH_SYSTEM_PAGE_PRIORITY_ACTIVE bug check has a value of 0x0000015B. This indicates that a worker thread's system page priority was leaked by the called worker routine.

IMPORTANT

This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

WORKER_THREAD_RETURNED_WITH_SYSTEM_PAGE_PRIORITY_ACTIVE Parameters

PARAMETER	DESCRIPTION
1	Address of worker routine (do ln on this address to find the offending driver)
2	Current system page priority value
3	WorkItem parameter
4	WorkItem address

Bug Check 0x15C: PDC_WATCHDOG_TIMEOUT_LIVEDUMP

7/2/2019 • 2 minutes to read • Edit Online

The PDC_WATCHDOG_TIMEOUT_LIVEDUMP bug check has a value of 0x0000015C. This indicates that a system component failed to respond within the allocated time period, preventing the system from entering or exiting connected standby.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

PDC_WATCHDOG_TIMEOUT_LIVEDUMP Parameters

PARAMETER	DESCRIPTION
1	Client ID of the hung component.
2	Client type of the hung component. 0x1 : A notification client failed to respond. 3 - Pointer to the notification client (pdc!_PDC_NOTIFICATION_CLIENT). 4 - Pointer to a pdc!PDC_14F_TRIAGE structure. 0x2 : A resiliency client failed to respond. 3 - Pointer to the resiliency client (pdc!_PDC_RESILIENCY_CLIENT). 4 - Pointer to a pdc!PDC_14F_TRIAGE structure. 0x3 : An activator client held a reference for too long 3 - Pointer to the activation client (pdc!_PDC_ACTIVATOR_CLIENT). 4 - Pointer to a pdc!PDC_14F_TRIAGE structure.
3	See parameter 2
4	See parameter 2

Bug Check 0x15D: SOC_SUBSYSTEM_FAILURE_LIVEDUMP

10/23/2019 • 2 minutes to read • Edit Online

The SOC_SUBSYSTEM_FAILURE_LIVEDUMP bug code has a value of 0x0000015D. This indicates that a System on a Chip (SoC) subsystem has experienced a critical fault and has captured a live kernel dump. The SoC subsystem does not generate a bug check in this situation.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Bug Check 0x14B SOC_SUBSYSTEM_FAILURE Parameters

PARAMETER	DESCRIPTION
1	Address of an **SOC_SUBSYSTEM_FAILURE_DETAILS** structure.
2	Reserved.
3	Reserved.
4	Optional. Address of a vendor-supplied data block.

Resolution

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause.

Use the provided nt!SOC_SUBSYSTEM_FAILURE_DETAILS structure to dump the failure data using the dt command and the address provided by Arg1.

```
2: kd> dt nt!SOC_SUBSYSTEM_FAILURE_DETAILS 9aa8d630
   +0x000 SubsysType       : 1 ( SOC_SUBSYS_AUDIO_DSP )
   +0x008 FirmwareVersion  : 0
   +0x010 HardwareVersion  : 0
   +0x018 UnifiedFailureRegionSize : 0x24
   +0x01c UnifiedFailureRegion : [1]  "F"
```

Work with SoC vendor to further parse the data, including the optional vendor supplied general purpose data block.

You may want to examine the stack trace using the **k, kb, kc, kd, kp, kP, kv (Display Stack Backtrace)** command. You can specify the processor number to examine the stacks on all processors.

You can also set a breakpoint in the code leading up to this stop code and attempt to single step forward into the faulting code.

For more information see the following topics:

Crash dump analysis using the Windows debuggers (WinDbg)

If you are not equipped to use the Windows debugger to work on this problem, you can use some basic troubleshooting techniques.

- Check the System Log in Event Viewer for additional error messages that might help identify the device or driver that is causing this bug check.
- If a driver is identified in the bug check message, disable the driver or check with the manufacturer for driver updates.
- You can try running the hardware diagnostics supplied by the system manufacturer.
- For additional general troubleshooting information, see **Blue Screen Data**.

Requirements

Minimum supported client	Windows 8
Minimum supported server	Windows Server 2012

Bug Check 0x15E: BUGCODE_NDIS_DRIVER_LIVE_DUMP

10/24/2019 • 4 minutes to read • Edit Online

The BUGCODE_NDIS_DRIVER_LIVE_DUMP bug code has a value of 0x0000015E. This bug code indicates that NDIS has captured a live kernel dump. NDIS does not generate a bug check in this situation.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

BUGCODE_NDIS_DRIVER Parameters

Parameter 1 indicates the type of violation. The meaning of the other parameters depends on the value of Parameter 1. If a Parameter's value is "0," that means it is not used.

PARAMETER 1	PARAMETER 1 VALUE AND CAUSE OF ERROR	PARAMETER 2	PARAMETER 3	PARAMETER 4
0x01	NDIS_BUGCHECK_MINIPORT_FATAL_ERROR A miniport driver has encountered a fatal error and requested re-enumeration.	The address of the miniport block. Run **!ndiskd.minidriver** with this address for more information.	The address of the miniport's Physical Device Object (PDO)	The fatal error that caused this live dump to be taken. Possible values: 1. 70: Caused by user mode 2. 71: Caused by **NdisMRemoveMiniport** 3. 72: Caused by **NdisIMInitializeDeviceInstanceEx** failing 4. 73: Caused by *MiniportRestart* failing 5. 74: Caused by failing a OID_PNP_SET_POWER (D0) request 6. 75: Caused by failing a OID_PNP_SET_POWER (Dx) request
0x25	NDIS_BUGCHECK_WATCHDOG An attempt to manage the	The operation that took too long. Possible values:	Cast to ndis!NDIS_WATCHDOG_TRIAGE_BLOCK. Useful	The value of Parameter 4 depends on the value of

PARAMETER 1	PARAMETER 1 VALUE AND CAUSE OF ERROR	PARAMETER 2	PARAMETER 3	PARAMETER 4
	network stack has taken too long. When NDIS calls out into other drivers, NDIS starts a watchdog timer to ensure the call completes promptly. If the call takes too long, NDIS injects a bugcheck. This can be caused by a simple deadlock. Look with "!stacks 2 ndis" or similar to see if any threads look suspicious. Pay special attention to the PrimaryThread from the NDIS_WATCHDOG_TRIAGE_BLOCK. This can be caused by lost NBLs, in which case **!ndiskd.pendingnbls** may help. Check for OIDs that are stuck using **!ndiskd.oid**.	• 0x01 : NDIS_BUGCHECK_WATCHDOG_PROTOCOL_PAUSE There was a timeout while pausing a protocol driver. • 0x02 : NDIS_BUGCHECK_WATCHDOG_PROTOCOL_NETPNPEVENT There was a timeout while delivering a NET_PNP_EVENT_NOTIFICATION to a protocol driver. • 0x03 : NDIS_BUGCHECK_WATCHDOG_PROTOCOL_STATUS_INDICATION There was a timeout while delivering a status indication to a protocol driver. • 0x04 : NDIS_BUGCHECK_WATCHDOG_PROTOCOL_UNBIND There was a timeout while unbinding a protocol driver.	fields: • **StartTime** shows what time the operation started, in 100ns units, as returned by KeQueryInterruptTime. • **TimeoutMilliseconds** shows how long NDIS waited, at a minimum, before triggering this bugcheck. • **TargetObject** is a handle to the protocol, filter module, or miniport adapter that NDIS is waiting on. Run **!ndiskd.protocol**, **!ndiskd.filter**, or **!ndiskd.netadapter** with this handle for more information. • **PrimaryThread** is the thread on which NDIS initiated the operation. Usually, this is the first place to look, although the thread may have gone elsewhere if the operation is being handled asynchronously.	Parameter 2. Each number in this list corresponds to the same number in Parameter 2. • 0x01 : 0 • 0x02 : The NET_PNP_EVENT_CODE of the stuck event. For more information about these codes, see **NET_PNP_EVENT**.. • 0x03 : The NDIS_STATUS code of the stuck indication. Use **!ndiskd.help** to decode it. • 0x04 : 0 • 0x11 : 0 • 0x12 : The NET_PNP_EVENT_CODE of the stuck event. For possible values, see the previous list of values for item 2 in this list. • 0x13 : The NDIS_STATUS code of the stuck indication. Use **!ndiskd.help** to decode it. • 0x14 : 0 • 0x21 : 0 • 0x22 : 0 • 0x23 : The OID code of the stuck request. Use **!ndiskd.help** to decode it. • 0x24 : The OID code of the stuck request. Use **!ndiskd.help** to decode it. • 0x25 : 0 • 0x26 : 0

PARAMETER 1	PARAMETER 1 VALUE AND CAUSE OF ERROR	PARAMETER 2	PARAMETER 3	PARAMETER 4
		• 0x11 : NDIS_BUGCHECK_WATCHDOG_FILTER_PAUSE There was a timeout while pausing a filter driver. • 0x12 : NDIS_BUGCHECK_WATCHDOG_FILTER_NETPNPEVENT There was a timeout while delivering a NET_PNP_EVENT_NOTIFICATION to a filter driver. • 0x13 : NDIS_BUGCHECK_WATCHDOG_FILTER_STATUS_INDICATION There was a timeout while delivering a status indication to a filter driver. • 0x14 : NDIS_BUGCHECK_WATCHDOG_FILTER_DETACH There was a timeout while detaching a filter driver. • 0x21 : NDIS_BUGCHECK_W		

PARAMETER 1	PARAMETER 1 VALUE AND CAUSE OF ERROR	PARAMETER 2	PARAMETER 3	PARAMETER 4
		ATCHDOG_MINIPORT_PAUSE There was a timeout while pausing a miniport adapter. • 0x22 : NDIS_BUGCHECK_WATCHDOG_MINIPORT_HALT There was a timeout while halting a miniport adapter. • 0x23 : NDIS_BUGCHECK_WATCHDOG_MINIPORT_OID There was a timeout while delivering an OID request to a miniport adapter. • 0x24 : NDIS_BUGCHECK_WATCHDOG_FILTER_OID There was a timeout while delivering an OID request to a filter driver. • 0x25 : NDIS_BUGCHECK_WATCHDOG_MINIPORT_IDLE There was a timeout while idling a miniport		

PARAMETER 1	PARAMETER 1 VALUE AND CAUSE OF ERROR	PARAMETER 2	PARAMETER 3	PARAMETER 4
		adapter. • 0x26 : NDIS_BUGCHECK_WATCHDOG_CANCEL_IDLE There was a timeout while canceling an idle request on a miniport adapter.		
0x30	NDIS_BUGCHECK_STUCK_NBL A miniport driver has not returned a NBL back to the stack for some time.	The address of the miniport block. Run **!ndiskd.minidriver** with this address for more information.	0	0

Cause

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause. Parameter 1 indicates the specific cause of the BUGCODE_NDIS_DRIVER_LIVE_DUMP bugcheck.

Remarks

NDIS has detected and recovered from a serious problem in another network driver. Although the system was not halted, this problem may later cause connectivity problems or a fatal bugcheck.

This bug code occurs only in Windows 8.1 and later versions of Windows.

Bug Check 0x15F: CONNECTED_STANDBY_WATCHDOG_TIMEOUT_LIVEDUMP

7/2/2019 • 2 minutes to read • Edit Online

The CONNECTED_STANDBY_WATCHDOG_TIMEOUT_LIVEDUMP bug check has a value of 0x0000015F. This indicates that a connected standby watchdog timeout has occurred.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

CONNECTED_STANDBY_WATCHDOG_TIMEOUT_LIVEDUMP Parameters

PARAMETER	DESCRIPTION
1	CS watchdog subcode 0x1 : DRIPS watchdog timeout. The system has been in the resiliency phase of connected standby with no activators active and no device constraints unsatisfied for too long without entering DRIPS (deepest runtime idle platform state). 2 - A pointer to additional information (nt!POP_DRIPS_WATCHDOG_METRICS) 3 - Non-DRIPS duration in milliseconds 4 - Reserved 0x2 : DRIPS watchdog device constraint timeout. The system has been in the resiliency phase of connected standby for too long without entering DRIPS (deepest runtime idle platform state) due to an unsatisfied device constraint with no activators active. 2 - nt!TRIAGE_POP_FX_DEVICE Device 3 - Component index 4 - Reserved 0x3 : DRIPS watchdog preveto timeout. The system has been in the resiliency phase of connected standby for too long without entering DRIPS (deepest runtime idle platform state) due to an active PEP pre-veto with no unsatisfied device constraint and with no activators active. 2 - Veto code 3 - A pointer to the veto name string (PWSTR) 4 - A pointer to the PEP PPM callback 0x4 : Deep Sleep Watchdog 2 - Metrics 3 -NonDeepSleepDurationMs 4 - Reserved 0x5 : Deep Sleep Power Setting Watchdog 2 - Metrics 3 -NonDeepSleepDurationMs 4 - Reserved

PARAMETER	DESCRIPTION
2	See parameter 1
3	See parameter 1
4	See parameter 1

Cause

This machine is exhibiting behavior that reduces screen-off battery life. Typically this is caused by CPU activity, device activity, or devices being in an insufficient idle state.

Bug Check 0x160: WIN32K_ATOMIC_CHECK_FAILURE

7/2/2019 • 2 minutes to read • Edit Online

The WIN32K_ATOMIC_CHECK_FAILURE bug check has a value of 0x00000160. This indicates that a Win32k function has violated an ATOMICCHECK.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

WIN32K_ATOMIC_CHECK_FAILURE Parameters

PARAMETER	DESCRIPTION
1	Count of functions on the stack currently inside of an ATOMIC operation
2	Reserved
3	Reserved
4	Reserved

Bug Check 0x161: LIVE_SYSTEM_DUMP

7/2/2019 • 2 minutes to read • Edit Online

The LIVE_SYSTEM_DUMP bug check has a value of 0x00000161. This indicates that the system administrator requested the collection of a live system memory dump.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

LIVE_SYSTEM_DUMP Parameters

None

Bug Check 0x162: KERNEL_AUTO_BOOST_INVALID_LOCK_RELEASE

7/2/2019 • 2 minutes to read • Edit Online

The KERNEL_AUTO_BOOST_INVALID_LOCK_RELEASE bug check has a value of 0x00000162. This indicates that a lock tracked by AutoBoost was released by a thread that did not own the lock.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

KERNEL_AUTO_BOOST_INVALID_LOCK_RELEASE Parameters

PARAMETER	DESCRIPTION
1	The address of the thread
2	The lock address
3	The session ID of the thread
4	Reserved

Cause

This is typically caused when some thread releases a lock on behalf of another thread (which is not legal with AutoBoost tracking enabled) or when some thread tries to release a lock it no longer owns.

Bug Check 0x163: WORKER_THREAD_TEST_CONDITION

7/2/2019 • 2 minutes to read • Edit Online

The WORKER_THREAD_TEST_CONDITION bug check has a value of 0x00000163. This indicates that a test for kernel worker threads raised a failure.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

WORKER_THREAD_TEST_CONDITION Parameters

PARAMETER	DESCRIPTION
1	Active test flags
2	Flag corresponding to the test that triggered the failure
3	Reserved
4	Reserved

Bug Check 0x164: WIN32K_CRITICAL_FAILURE

7/2/2019 • 2 minutes to read • Edit Online

The WIN32K_CRITICAL_FAILURE bug check has a value of 0x00000164. This indicates that Win32k has encountered a critical failure.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

WIN32K_CRITICAL_FAILURE Parameters

PARAMETER	DESCRIPTION
1	1 - Type of the failure. 0x1 : REGION_VALIDATION_FAILURE- Region is out of surface bounds. 2 - Pointer to DC 3 - Pointer to SURFACE 4 - Pointer to REGION 0x2 : OPERATOR_NEW_USED - Operator "new" is used to allocate memory. 2 - Reserved 3 - Reserved 4 - Reserved 0x3 : CRITICAL_APISET_EXTENSIONS_MISSING - Critical extension APISET API is missing. 2 - wchar_t* to the name of the missing function 3 - Reserved 4 - Reserved 0x4 : GDI_SPRITE_SURFACE_INVALID_DELETE - GDI sprite's shape is being deleted without deleting the sprite. 2 - Handle to the SURFACE 3 - Reference count to the SURFACE 4 - PID of the SURFACE owner 0x5 : POINTER_DEVICE_EXCLUSIVE_OPEN_FAILED - Failed to open Pointer device. 2 - UNICODE_STRING of the device 3 - Reserved 4 - Reserved 0x8 : PUBLIC_DC_INVALID_PRIVATE_MEMBER - A public DC has a pointer to an object owned by a specific process. 2 - Pointer to DC

PARAMETER	DESCRIPTION
	3 - Process id that owns the object 4 - Reserved 0xA : TTFD_INVOKE_ILLEGAL_ID - Invalid function table index is being used in TTFD. 2 - Reserved 3 - Reserved 4 - Reserved 0xB : OTFD_INVOKE_ILLEGAL_ID - Invalid function table index is being used in ATMFD. 2 - Reserved 3 - Reserved 4 - Reserved 0xC : GFPE_INVOKE_ILLEGAL_ID - Invalid function table index is being used in a PALETTE. 2 - Pointer to the PALETTE 3 - The invalid index 4 - Maximum valid index + 1 0x10 : USER_SAS_REGISTRATION_FAILED - SAS key registration has failed. 2 - vkey 3 - modifiers 4 - flags
2	See parameter 1
3	See parameter 1
4	See parameter 1

Bug Check 0x165: CLUSTER_CSV_STATUS_IO_TIMEOUT_LIVEDUMP

7/2/2019 • 2 minutes to read • Edit Online

The CLUSTER_CSV_STATUS_IO_TIMEOUT_LIVEDUMP bug check has a value of 0x00000165. This indicates that a SMB client on the non-coordinating node complains that an IO on coordinating node is taking too long and fails all IOs with STATUS_IO_TIMEOUT.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

CLUSTER_CSV_STATUS_IO_TIMEOUT_LIVEDUMP Parameters

PARAMETER	DESCRIPTION
1	Optional cluster service PID.
2	Cluster Node Id for the node that observed STATUS_IO_TIMEOUT.
3	Reserved.
4	Reserved.

Cause

A SMB client on the non-coordinating node complains that an IO on coordinating node is taking too long and fails all IOs with STATUS_IO_TIMEOUT.

Additional information is available in the dump's secondary data streams.

(This code can never be used for a real bugcheck; it is used to identify live dumps including Cluster Shared Volume telemetry.)

Resolution

See Also

Troubleshooting Hangs Using Live Dump (Blog)

Bug Check Code Reference

Bug Check 0x166: CLUSTER_RESOURCE_CALL_TIMEOUT_LIVEDUMP

7/2/2019 • 2 minutes to read • Edit Online

The CLUSTER_RESOURCE_CALL_TIMEOUT_LIVEDUMP bug check has a value of 0x00000166. This indicates that a Cluster Resource call took longer than configured timeout.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

CLUSTER_RESOURCE_CALL_TIMEOUT_LIVEDUMP Parameters

PARAMETER	DESCRIPTION
1	Resource Host Monitor PID.
2	TID of the thread that handles resource call.
3	Resource call type - listed below.
4	Subcode. When parameter 3 equals 8 then this parameter contains cluster resource control code. When parameter 3 equals 9 then this parameter contains cluster resource type control code.

Resource call type

1 OPEN

2 CLOSE

3 ONLINE

4 OFFLINE

5 TERMINATE

6 ARBITRATE

7 RELEASE

8 RESOURCE CONTROL

9 RESOURCE TYPE CONTROL

10 LOOKS ALIVE

11 IS ALIVE

12 FAILURE NOTIFICATION

13 SHUTDOWN PROCESS

Cause

A Cluster Resource call took longer than configured timeout. The system generated a live dump for analysis of the delay.

(This code can never be used for a real bugcheck.)

Resolution

See Also

Troubleshooting Hangs Using Live Dump (Blog)

Bug Check Code Reference

Bug Check 0x167: CLUSTER_CSV_SNAPSHOT_DEVICE_INFO_TIMEOUT_LIVEDUMP

7/2/2019 • 2 minutes to read • Edit Online

The CLUSTER_CSV_SNAPSHOT_DEVICE_INFO_TIMEOUT_LIVEDUMPP bug check has a value of 0x00000167. This indicates that a Cluster Service call to the volsnap to query snapshot information took too long.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

CLUSTER_CSV_SNAPSHOT_DEVICE_INFO_TIMEOUT_LIVEDUMP Parameters

PARAMETER	DESCRIPTION
1	Cluster Service PID.
2	TID of the thread that handles volsnap query.
3	This parameter has a value of 1 if we timeout out while CSV volume is active and 2 if we timeout out even after taking CSV volume down.
4	Reserved.

Cause

A Cluster Service call to the volsnap to query snapshot information took too long.

(This code can never be used for a real bugcheck.)

Resolution

See Also

Troubleshooting Hangs Using Live Dump (Blog)

Bug Check Code Reference

Bug Check 0x168: CLUSTER_CSV_STATE_TRANSITION_TIMEOUT_LIVEDUMP

7/2/2019 • 2 minutes to read • Edit Online

The CLUSTER_CSV_STATE_TRANSITION_TIMEOUT_LIVEDUMP bug check has a value of 0x00000168. This indicates that a Cluster Shared Volume state transition took too long.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

CLUSTER_CSV_STATE_TRANSITION_TIMEOUT_LIVEDUMP Parameters

PARAMETER	DESCRIPTION
1	Cluster Service PID.
2	CSV target state Id - listed below.
3	Reserved
4	Reserved

CSV target state Id

0 Waiting for volume to transition to the Init state.

1 Waiting for volume to transition to the Paused state.

2 Waiting for volume to transition to the Draining state.

3 Waiting for volume to transition to the Set-Down-Level state.

4 Waiting for volume to transition to the Active state.

Cause

A Cluster Shared Volume state transition took too long. The system generated a live dump for analysis of the delay.

(This code can never be used for a real bugcheck.)

Resolution

See Also

Troubleshooting Hangs Using Live Dump (Blog)

Bug Check Code Reference

Bug Check 0x169: CLUSTER_CSV_VOLUME_ARRIVAL_LIVEDUMP

7/2/2019 • 2 minutes to read • Edit Online

The CLUSTER_CSV_VOLUME_ARRIVAL_LIVEDUMP bug check has a value of 0x00000169. This indicates that Cluster Shared Volume Manager was asked to create a new volume device object, and volume has not arrived in time.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

CLUSTER_CSV_VOLUME_ARRIVAL_LIVEDUMP Parameters

PARAMETER	DESCRIPTION
1	Cluster Service PID.
2	Reserved.
3	Reserved.
4	Reserved.

Cause

The Cluster Shared Volume Manager was asked to create a new volume device object, and volume has not arrived in time.

The system generated a live dump for analysis of the delay.

(This code can never be used for a real bugcheck.)

Resolution

See Also

Troubleshooting Hangs Using Live Dump (Blog)

Bug Check Code Reference

Bug Check 0x16A: CLUSTER_CSV_VOLUME_REMOVAL_LIVEDUMP

7/2/2019 • 2 minutes to read • Edit Online

The CLUSTER_CSV_VOLUME_REMOVAL_LIVEDUMP bug check has a value of 0x0000016A. This indicates that a Cluster Shared Volume Manager volume removal request has timed out.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

CLUSTER_CSV_VOLUME_REMOVAL_LIVEDUMP Parameters

PARAMETER	DESCRIPTION
1	Cluster Service PID
2	Reserved.
3	Reserved.
4	Reserved.

Cause

A Cluster Shared Volume Manager volume removal request has timed out.

The system generated a live dump for analysis of the delay.

(This code can never be used for a real bugcheck.)

Resolution

See Also

Troubleshooting Hangs Using Live Dump (Blog)

Bug Check Code Reference

Bug Check 0x16B: CLUSTER_CSV_CLUSTER_WATCHDOG_LIVEDUMP

7/2/2019 • 2 minutes to read • Edit Online

The CLUSTER_CSV_CLUSTER_WATCHDOG_LIVEDUMP bug check has a value of 0x0000016B. This indicates that the Cluster service user mode watchdog detected that a thread is not making forward progress for a long time.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

CLUSTER_CSV_CLUSTER_WATCHDOG_LIVEDUMP Parameters

PARAMETER	DESCRIPTION
1	Cluster Service PID.
2	Id of the thread that is stuck.
3	Reserved.
4	Reserved.

Cause

The Cluster service user mode watchdog detected that a thread is not making forward progress for a long time.

The system generated a live dump for analysis of the delay.

(This code can never be used for a real bugcheck.)

Resolution

See Also

Troubleshooting Hangs Using Live Dump (Blog)

Bug Check Code Reference

Bug Check 0x16C: INVALID_RUNDOWN_PROTECTION_FLAGS

7/2/2019 • 2 minutes to read • Edit Online

The INVALID_RUNDOWN_PROTECTION_FLAGS bug check has a value of 0x0000016C. This indicates that the flags supplied to one of the rundown protection APIs were invalid.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

INVALID_RUNDOWN_PROTECTION_FLAGS Parameters

PARAMETER	DESCRIPTION
1	The invalid flags supplied by the caller
2	The address of the rundown ref
3	Reserved
4	Reserved

Bug Check 0x16D: INVALID_SLOT_ALLOCATOR_FLAGS

7/2/2019 • 2 minutes to read • Edit Online

The INVALID_SLOT_ALLOCATOR_FLAGS bug check has a value of 0x0000016D. This indicates that the flags supplied to one of the slot allocator APIs were invalid.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

INVALID_SLOT_ALLOCATOR_FLAGS Parameters

PARAMETER	DESCRIPTION
1	The invalid flags supplied by the caller
2	Reserved
3	Reserved
4	Reserved

Bug Check 0x16E: ERESOURCE_INVALID_RELEASE

7/2/2019 • 2 minutes to read • Edit Online

The ERESOURCE_INVALID_RELEASE bug check has a value of 0x0000016E. This indicates that the target thread pointer supplied to ExReleaseResourceForThreadLite was invalid.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

ERESOURCE_INVALID_RELEASE Parameters

PARAMETER	DESCRIPTION
1	The resource being released
2	The current thread
3	The incorrect target thread that was passed in
4	Reserved

Cause

This bugcheck will hit if a call to ExSetOwnerPointerEx was skipped by the API client (if a cross-thread release was intended) or if the caller accidentally passed in a value other that supplied by ExGetCurrentResourceThread.

Bug Check 0x16F: CLUSTER_CSV_STATE_TRANSITION_INTERVAL_TIMEOUT_LIVEDUMP

7/2/2019 • 2 minutes to read • Edit Online

The CLUSTER_CSV_STATE_TRANSITION_INTERVAL_TIMEOUT_LIVEDUMP bug check has a value of 0x0000016F. This indicates that a Cluster Shared Volume next state transition request has not arrived.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

CLUSTER_CSV_STATE_TRANSITION_INTERVAL_TIMEOUT_LIVEDUMP Parameters

PARAMETER	DESCRIPTION
1	Cluster Service PID.
2	CSV target state Id - listed below.
3	Reserved
4	Reserved

CSV target state Id

0 Waiting for volume to transition to the Init state.

1 Waiting for volume to transition to the Paused state.

2 Waiting for volume to transition to the Draining state.

3 Waiting for volume to transition to the Set-Down-Level state.

4 Waiting for volume to transition to the Active state.

Cause

The Cluster Shared Volume next state transition request has not arrived.

The system generated a live dump for analysis of the delay.

(This code can never be used for a real bugcheck.)

Resolution

See Also

Troubleshooting Hangs Using Live Dump (Blog)

Bug Check Code Reference

Bug Check 0x170: CRYPTO_LIBRARY_INTERNAL_ERROR

7/2/2019 • 2 minutes to read • Edit Online

The CRYPTO_LIBRARY_INTERNAL_ERROR bug check has a value of 0x00000170. It indicates that an internal error in the crypto libraries occurred.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

CRYPTO_LIBRARY_INTERNAL_ERROR Parameters

PARAMETER	DESCRIPTION
1	ID of failure.
2	Reserved.
3	Reserved.
4	Reserved.

Cause

This bugcheck indicates the cryptographic library hit an anomaly which should never occur, and the library has no safe method of signaling the error to the caller. This might be the symptom of an active attack.

See Also

Bug Check Code Reference

Cryptography API: Next Generation

Bug Check 0x171: CLUSTER_CSV_CLUSSVC_DISCONNECT_WATCHDOG

7/2/2019 • 2 minutes to read • Edit Online

The CLUSTER_CSV_CLUSSVC_DISCONNECT_WATCHDOG bug check has a value of 0x00000171. This indicates that Cluster disconnect is not making forward progress.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

CLUSTER_CSV_CLUSSVC_DISCONNECT_WATCHDOG Parameters

PARAMETER	DESCRIPTION
1	Id of the thread that is handling cluster disconnect.
2	Timeout in milliseconds.
3	Reserved
4	Reserved

Cause

The Cluster disconnect is not making forward progress.

Resolution

See Also

Troubleshooting Hangs Using Live Dump (Blog)

Bug Check Code Reference

Bug Check 0x173: COREMSGCALL_INTERNAL_ERROR

7/2/2019 • 2 minutes to read • Edit Online

The PFN_SHARE_COUNT bug check has a value of 0x00000173. This indicates that the CoreMessageCall detected an unrecoverable error.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

COREMSGCALL_INTERNAL_ERROR Parameters

PARAMETER	DESCRIPTION
1	Type of the failure.
2	Reserved.
3	Reserved.
4	Reserved.

Cause

CoreMessageCall detected an unrecoverable error.

See Also

Bug Check Code Reference

Bug Check 0x174: COREMSG_INTERNAL_ERROR

7/2/2019 • 2 minutes to read • Edit Online

The COREMSG_INTERNAL_ERROR bug check has a value of 0x00000174. This indicates that CoreMessaging detected an unrecoverable error.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

COREMSG_INTERNAL_ERROR Parameters

PARAMETER	DESCRIPTION
1	Type of the failure.
2	Reserved.
3	Reserved.
4	Reserved.

Cause

CoreMessaging detected an unrecoverable error.

See Also

Bug Check Code Reference

Bug Check 0x175: PREVIOUS_FATAL_ABNORMAL_RESET_ERROR

7/2/2019 • 2 minutes to read • Edit Online

The PREVIOUS_FATAL_ABNORMAL_RESET_ERROR bug check has a value of 0x00000175. This indicates that an unrecoverable system error occurred or the system has abnormally reset on Windows phone devices. The system generated a live dump to collect device crash data from the previous error.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

PREVIOUS_FATAL_ABNORMAL_RESET_ERROR Parameters

PARAMETER	DESCRIPTION
1	Reserved
2	Reserved
3	Reserved
4	Reserved

Cause

The system on Windows Phone devices encountered an unexpected error and restarted. Issues that may cause this error include: hardware watchdog timer in application or auxiliary processors indicating a system hang, user-initiated key sequence because of a hang, etc.

Bug Check 0x178: ELAM_DRIVER_DETECTED_FATAL_ERROR

7/2/2019 • 2 minutes to read • Edit Online

The ELAM_DRIVER_DETECTED_FATAL_ERROR bug check has a value of 0x00000178. This indicates that ELAM driver detected a fatal error.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

ELAM_DRIVER_DETECTED_FATAL_ERROR Parameters

PARAMETER	DESCRIPTION
1	Type of the failure. 0x0 : TPM attestation could not be revoked 2 - Pointer to the BDCB_IMAGE_INFORMATION structure for the driver being inspected 3 - TBS_RESULT failure code 0x10000 : ELAM-vendor defined failure 2 - (Optional) ELAM vendor supplied value 3 - (Optional) ELAM vendor supplied value
2	See parameter 1
3	See parameter 1
4	(Optional) ELAM vendor supplied general purpose data block

Bug Check 0x179: CLUSTER_CLUSPORT_STATUS_IO_TIMEOUT_LIVEDUMP

7/2/2019 • 2 minutes to read • Edit Online

The CLUSTER_CLUSPORT_STATUS_IO_TIMEOUT_LIVEDUMP bug check has a value of 0x00000179. This indicates that SMB client on the initiator node complains that an IO to a target node is taking too long and fails all IOs with STATUS_IO_TIMEOUT.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

CLUSTER_CLUSPORT_STATUS_IO_TIMEOUT_LIVEDUMP Parameters

PARAMETER	DESCRIPTION
1	IRP that ran into the timeout.
2	Reserved.
3	Reserved.
4	Reserved.

Cause

SMB client on the initiator node complains that an IO on target node is taking too long and fails all IOs with STATUS_IO_TIMEOUT.

In response to that, the cluster initiator captures live dump on the initiator node so we can analyze what IO is taking time.

Additional information is available in the dump's secondary data streams.

(This code can never be used for a real bugcheck.)

Resolution

See Also

Troubleshooting Hangs Using Live Dump (Blog)

Bug Check Code Reference

Bug Check Bug Check 0x17B: PROFILER_CONFIGURATION_ILLEGAL

7/2/2019 • 2 minutes to read • Edit Online

The PROFILER_CONFIGURATION_ILLEGAL bug check has a value of 0x0000017B.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

PROFILER_CONFIGURATION_ILLEGAL Parameters

None

Bug Check 17C: PDC_LOCK_WATCHDOG_LIVEDUMP

7/2/2019 • 2 minutes to read • Edit Online

The PDC_LOCK_WATCHDOG_LIVEDUMP bug check has a value of 0x0000017C. This indicates that a thread has been holding the PDC lock for too long.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

PDC_LOCK_WATCHDOG_LIVEDUMP Parameters

PARAMETER	DESCRIPTION
1	The thread holding the PDC lock.
2	Lock watchdog timeout in milliseconds.
3	Reserved.
4	Reserved.

Cause

A thread has been holding the PDC lock for too long. A livedump is created to provide information to investigate.

(This code can never be used for a real bugcheck.)

Resolution

Use the debugger !thread command to display the thread holding the lock that is provided in parameter 1. Analyze that code to determine why it is holding the lock beyond the timeout period.

See Also

Bug Check Code Reference

!thread

Bug Check 0x17D: PDC_UNEXPECTED_REVOCATION_LIVEDUMP

7/2/2019 • 2 minutes to read • Edit Online

The PDC_UNEXPECTED_REVOCATION_LIVEDUMP bug check has a value of 0x0000017D. It indicates that an activator has been revoked unexpectedly.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

PDC_UNEXPECTED_REVOCATION_LIVEDUMP Parameters

PARAMETER	DESCRIPTION
1	The client ID of the revoked activator.
2	The revoked activator client.
3	The revoked activation instance.
4	pdc!_PDC_CLIENT_PROCESS_INFO

Cause

An activator has been revoked unexpectedly.

A livedump is created to provide information to investigate.

(This code can never be used for a real bugcheck.)

See Also

Bug Check Code Reference

Bug Check 0x17E: MICROCODE_REVISION_MISMATCH

7/2/2019 • 2 minutes to read • Edit Online

The MICROCODE_REVISION_MISMATCH bug check has a value of 0x0000017E. It indicates that one or more processors in the multiprocessor configuration have inconsistent microcode loaded.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

MICROCODE_REVISION_MISMATCH Parameters

PARAMETER	DESCRIPTION
1	The processor CPUID signature value of the processor that mismatched.
2	The expected microcode revision for the processor.
3	The actual, reported microcode revision for the processor.
4	The processor index of the mismatching processor.

Cause

One or more processors in the multiprocessor configuration have inconsistent microcode loaded. This bugcheck indicates that faulty system firmware has mistakenly applied a microcode update to only a subset of processors in the host configuration. System firmware must apply microcode updates to all processors in a uniform fashion.

See Also

Bug Check Code Reference

Bug Check 0x187: VIDEO_DWMINIT_TIMEOUT_FALLBACK_BDD

7/2/2019 • 2 minutes to read • Edit Online

The VIDEO_DWMINIT_TIMEOUT_FALLBACK_BDD bug check has a value of 0x00000187. This indicates that video fell back to BDD rather than using the IHV driver. This always generates a live dump.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

VIDEO_DWMINIT_TIMEOUT_FALLBACK_BDD Parameters

PARAMETER	DESCRIPTION
1	Reason Code. 0x1 : DWM failed to initialize after retries, stopping display adapters and falling back to BDD
2	Reserved
3	Reserved
4	Reserved

Bug Check 0x188: CLUSTER_CSVFS_LIVEDUMP

7/2/2019 • 2 minutes to read • Edit Online

The CLUSTER_CSVFS_LIVEDUMP bug check has a value of 0x00000188. This indicates that CSVFS initiated this livedump to help debug an inconsistent state.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

CLUSTER_CSVFS_LIVEDUMP Parameters

PARAMETER	DESCRIPTION
1	Reason Code 0x1 : Cache purge on oplock downgrade to none has failed 2- address of CSVFS!_SCB 0x2 : Cache purge on oplock upgrade from none has failed 2- address of CSVFS!_SCB 0x3 : Cache purge on set purge failure mode 2- address of CSVFS!_SCB 0x4 : Cache flush on oplock downgrade to none failed 2- address of CSVFS!_SCB
2	See parameter 1
3	Reserved
4	Reserved

Cause

First parameter contains the reason code When CSVFS detects that current state might cause data corruption or other sort of inconsistency it would generate live dump with this status code. Parameter1 has code pointing to what scenario this live dump is created for. Other parameters should be interpreted in context of the reason code.

Bug Check 0x189: BAD_OBJECT_HEADER

7/2/2019 • 2 minutes to read • Edit Online

The BAD_OBJECT_HEADER bug check has a value of 0x00000189. This indicates that The OBJECT_HEADER has been corrupted.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

BAD_OBJECT_HEADER Parameters

PARAMETER	DESCRIPTION
1	Pointer to bad OBJECT_HEADER
2	Pointer to the resulting OBJECT_TYPE based on the TypeIndex in the OBJECT_HEADER
3	Type of corruption. 0x0 : The type index is corrupt 0x1 : The object security descriptor is invalid
4	Reserved

Bug Check 0x18B: SECURE_KERNEL_ERROR

7/2/2019 • 2 minutes to read • Edit Online

The SECURE_KERNEL_ERROR bug check has a value of 0x0000018B. This indicates that the secure kernel has encountered a fatal error.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

SECURE_KERNEL_ERROR Parameters

PARAMETER	DESCRIPTION
1	Reserved
2	Reserved
3	Reserved
4	Reserved

Bug Check 0x18C: HYPERGUARD_VIOLATION

7/2/2019 • 2 minutes to read • Edit Online

The HYPERGUARD_VIOLATION bug check has a value of 0x0000018C. This indicates that the kernel has detected that critical kernel code or data have been corrupted.

IMPORTANT

This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

NOTE

This bug code is reserved for use by Hyperguard only.
It is not a general purpose bug code intended for use by other components in data corruption scenarios.
Instead, define a unique bug code for your component.
Do not use this bug code in your component.

HYPERGUARD_VIOLATION Parameters

PARAMETER	DESCRIPTION
1	Type of corrupted region - values listed below.
2	Failure type dependent information.
3	Reserved.
4	Reserved.

Type of corrupted region

1001 : A generic data region

1002 : A page hash mismatch

1004 : A processor IDT

1005 : A processor GDT

1007 : Debug routine modification

1008 : A dynamic code region

1009 : A generic shareable data region

100a : A hypervisor overlay region

100b : A processor mode misconfiguration

100c : An extended processor control register

100d : A secure memory region

100e : A loaded module

100f : A processor state region

1010 : The kernel CFG bitmap

1011 : The virtual address 0 page

1012 : The alternate inverted function table

1013 : An on-demand page verification failed

1016 : A secure image region

1017 : Kernel virtual address protection inconsistency

1101 : Internal context corruption

1102 : IDTR modification

1103 : GDTR modification

Cause

This bugcheck is generated when the kernel detects that critical kernel code or data have been corrupted. There are generally three causes for a corruption:

1. A driver has inadvertently or deliberately modified critical kernel code or data.
2. A developer attempted to set a normal kernel breakpoint using a kernel debugger that was not attached when the system was booted. Normal breakpoints, "bp", can only be set if the debugger is attached at boot time. Hardware breakpoints, "ba", can be set at any time.
3. A hardware corruption occurred, e.g. failing RAM holding kernel code or data.

See Also

Bug Check Code Reference

Bug Check 0x18D: SECURE_FAULT_UNHANDLED

7/2/2019 • 2 minutes to read • Edit Online

The SECURE_FAULT_UNHANDLED bug check has a value of 0x0000018D.

This bug check indidates that a secure fault originated by the secure kernel could not be handled.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

SECURE_FAULT_UNHANDLED Parameters

PARAMETER	DESCRIPTION
1	Secure fault code bitmask - values below.
2	Secure fault VA (only applicable to certain secure fault types).
3	Exception Record.
4	Context Record.

Secure fault code bitmask

```
0x1 : KSECURE_FAULT_SLAT_NX
    A no-execute fault occured due to SLAT page protections.
0x2 : KSECURE_FALT_SLAT_READ
    A read fault occurred due to SLAT page protections.
0x4 : KSECURE_FAULT_SLAT_WRITE
    A write fault occured due to SLAT page protections.
0x8 : KSECURE_FAULT_DOUBLE_FAULT
    A secure fault occurred before the prior secure fault had been dismissed by the kernel.
```

Cause

A secure fault originated by the secure kernel could not be handled.

See Also

Bug Check Code Reference

Bug Check Bug Check 0x18E: KERNEL_PARTITION_REFERENCE_VIOLATION

7/2/2019 • 2 minutes to read • Edit Online

The KERNEL_PARTITION_REFERENCE_VIOLATION bug check has a value of 0x0000018E.

This error indicates that a partition was improperly dereferenced. This normally occurs when a kernel-mode driver doesn't properly dereference partition objects. It can also occur when a serious data corruption occurs in the kernel.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

KERNEL_PARTITION_REFERENCE_VIOLATION Parameters

The following parameters are displayed on the blue screen.

Parameter 1 indicates the type of failure. The meaning of the other parameters depends on the value of Parameter 1.

PARAMETER 1	PARAMETER 2	PARAMETER 3	PARAMETER 4
0x0 : A partition with a non-zero hard reference count is being deleted.	Pointer to the partition.	Number of outstanding hard references.	Reserved
0x1 : The system partition is being deleted	Pointer to the partition.	Reserved	Reserved
0x2 : A partition with outstanding ex work queue items is being deleted.	Pointer to the partition.	Pointer to the ex work queue with outstanding items.	Reserved

Bug Check 0x190: WIN32K_CRITICAL_FAILURE_LIVEDUMP

7/2/2019 • 2 minutes to read • Edit Online

The WIN32K_CRITICAL_FAILURE_LIVEDUMP bug check has a value of 0x00000190. This indicates that Win32k has encountered a critical failure. A live dump is captured to collect the debug information.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

WIN32K_CRITICAL_FAILURE_LIVEDUMP Parameters

PARAMETER	DESCRIPTION
1	Type of the failure 0x1 : REGION_VALIDATION_FAILURE - Region is out of surface bounds. 2- Pointer to DC 3- Pointer to SURFACE 4- Pointer to REGION 0x2 : OPERATOR_NEW_USED - Operator "new" is used to allocate memory. 2 - Reserved 3 - Reserved 4 - Reserved
2	See parameter 1
3	See parameter 1
4	See parameter 1

Bug Check 0x191: PF_DETECTED_CORRUPTION

7/2/2019 • 2 minutes to read • Edit Online

The PF_DETECTED_CORRUPTION bug check has a value of 0x00000191.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

PF_DETECTED_CORRUPTION Parameters

PARAMETER	DESCRIPTION
1	Reserved
2	Reserved
3	Reserved
4	Reserved

Bug Check 0x192: KERNEL_AUTO_BOOST_LOCK_ACQUISITION_WITH_RAISED_IRQL

7/2/2019 • 2 minutes to read • Edit Online

The KERNEL_AUTO_BOOST_LOCK_ACQUISITION_WITH_RAISED_IRQL bug check has a value of 0x00000192. This indicates that a lock tracked by AutoBoost was acquired while executing at DISPATCH_LEVEL or above.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

KERNEL_AUTO_BOOST_LOCK_ACQUISITION_WITH_RAISED_IRQL Parameters

PARAMETER	DESCRIPTION
1	The address of the thread
2	The lock address
3	The IRQL at which the lock was acquired
4	Reserved

Cause

The caller cannot be blocking on a lock above APC_LEVEL because the lock may be held exclusively by the interrupted thread, which would cause a deadlock.

Bug Check 0x193: VIDEO_DXGKRNL_LIVEDUMP

7/2/2019 • 2 minutes to read • Edit Online

The VIDEO_DXGKRNL_LIVEDUMP bug check has a value of 0x00000193. This indicates a livedump triggered by dxgkrnl occurred.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

VIDEO_DXGKRNL_LIVEDUMP Parameters

PARAMETER	DESCRIPTION
1	Reason Code 0x100 Internal
2	Reserved
3	Reserved
4	Reserved

Resolution

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause.

Bug Check 0x195: SMB_SERVER_LIVEDUMP

7/2/2019 • 2 minutes to read • Edit Online

The SMB_SERVER_LIVEDUMP bug check has a value of 0x00000195. This indicates the SMB server detected a problem and has captured a kernel dump to collect debug information.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

SMB_SERVER_LIVEDUMP Parameters

PARAMETER	DESCRIPTION
1	0x1 : An I/O failed to complete in a reasonable amount of time. 2 - Pointer to the I/O's SRV2_WORK_ITEM
2	See parameter 1
3	Reserved
4	Reserved

Bug Check 0x196: LOADER_ROLLBACK_DETECTED

7/2/2019 • 2 minutes to read • Edit Online

The LOADER_ROLLBACK_DETECTED bug check has a value of 0x00000196. This indicates that the version of the OS loader does not match the operating system.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

LOADER_ROLLBACK_DETECTED Parameters

PARAMETER	DESCRIPTION
1	Loader security version
2	OS security version
3	Reserved
4	Reserved

Bug Check 0x197: WIN32K_SECURITY_FAILURE

7/2/2019 • 2 minutes to read • Edit Online

The WIN32K_SECURITY_FAILURE bug check has a value of 0x00000197. This indicates a security failure was detected in win32k.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

WIN32K_SECURITY_FAILURE Parameters

PARAMETER	DESCRIPTION
1	Failure type 0x1 : An objects handle entry didn't point back to the object. 2 - Pointer to the object type 3 - Pointer to the object handle entry 4 - Expected object
2	See parameter 1
3	See parameter 1
4	See parameter 1

Bug Check 0x198: UFX_LIVEDUMP

7/2/2019 • 2 minutes to read • Edit Online

The UFX_LIVEDUMP bug check has a value of 0x00000198. This indicates that a UFX live dump occurred.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

UFX_LIVEDUMP Parameters

PARAMETER	DESCRIPTION
1	Failure type 0x1 : A class driver failed to activate the bus. 2 - Mask of enumerated child PDOs 3 - Mask of activated child PDOs 4 - Reserved
2	See parameter 1
3	See parameter 1
4	See parameter 1

Bug Check 0x199: KERNEL_STORAGE_SLOT_IN_USE

7/2/2019 • 2 minutes to read • Edit Online

The KERNEL_STORAGE_SLOT_IN_USE bug check has a value of 0x00000199. This indicates that the storage slot cannot be freed because there is an object using it.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

KERNEL_STORAGE_SLOT_IN_USE Parameters

PARAMETER	DESCRIPTION
1	The address of the storage array
2	Reserved
3	Reserved
4	Reserved

Bug Check 0x19A: WORKER_THREAD_RETURNED_WHILE_ATTACHED_TO_SILO

7/2/2019 • 2 minutes to read • Edit Online

The WORKER_THREAD_RETURNED_WHILE_ATTACHED_TO_SILO bug check has a value of 0x0000019A. This indicates that a worker thread attached to a silo and did not detach before returning.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

WORKER_THREAD_RETURNED_WHILE_ATTACHED_TO_SILO Parameters

PARAMETER	DESCRIPTION
1	Address of worker routine
2	Workitem parameter
3	Workitem address
4	Reserved

Cause

To investigate use the **ln (List Nearest Symbols)**command on parameter 1 to help identify the mis-behaving driver.

Bug Check 0x19B: TTM_FATAL_ERROR

7/2/2019 • 2 minutes to read • Edit Online

The TTM_FATAL_ERROR bug check has a value of 0x0000019B. This indicates that the terminal topology manager experienced a fatal error.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

TTM_FATAL_ERROR Parameters

PARAMETER	DESCRIPTION
1	Failure type 0x1 : An terminal object could not be generated. 2 - The NT status code of the failure 3 - Reserved 4 - Reserved
2	See parameter 1
3	See parameter 1
4	See parameter 1

Bug Check 0x19C: WIN32K_POWER_WATCHDOG_TIMEOUT

7/2/2019 • 2 minutes to read • Edit Online

The WIN32K_POWER_WATCHDOG_TIMEOUT bug check has a value of 0x0000019C. This indicates that Win32k did not turn the monitor on in a timely manner.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

WIN32K_POWER_WATCHDOG_TIMEOUT Parameters

PARAMETER	DESCRIPTION
1	Failure type (win32kbase!POWER_WATCHDOG_TYPE) `0x10 : The power request queue is not making progress 2 - Pointer to the thread processing power requests, if any 3 - Pointer to the win32k user lock 4 - Pointer to the power request (win32kbase!PPOWERREQUEST) being processed, if any 0x20 : Calling PO to set power state 2 - Pointer to the power request worker thread 3 - Reserved 4 - Reserved 0x30 : Calling GDI to power on 2 - Pointer to the power request worker thread 3 - Reserved 4 - Reserved 0x40 : Calling DWM to render 2 - Pointer to the power request worker thread 3 - Reserved 4 - Reserved 0x50 : Calling monitor driver to power on 2 - Pointer to the power request worker thread 3 - Reserved 4 - Reserved`
2	See parameter 1
3	See parameter 1
4	See parameter 1

Resolution

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause.

Bug Check 0x19D: CLUSTER_SVHDX_LIVEDUMP

7/2/2019 • 2 minutes to read • Edit Online

The CLUSTER_SVHDX_LIVEDUMP bug check has a value of 0x0000019D. This indicates that SVHDX initiated this livedump to help debug an inconsistent state.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

CLUSTER_SVHDX_LIVEDUMP Parameters

PARAMETER	DESCRIPTION
1	Reason code 0x1 : Mounting a Shared Virtual disk has failed 2 - Address of Svhdxflt!_SVHDX_VIRTUALDISK_CONTEXT 3 - Address of nt!_FILE_OBJECT 4 - NTSTATUS
2	See parameter 1
3	See parameter 1
4	See parameter 1

Cause

When SVHDX detects that current state might cause some sort of inconsistency it will generate live dump with this status code. Parameter1 has code pointing to what scenario this live dump is created for. Other parameters should be interpreted in context of the reason code.

Bug Check 0x1A0: TTM_WATCHDOG_TIMEOUT

7/2/2019 • 2 minutes to read • Edit Online

The TTM_WATCHDOG_TIMEOUT bug check has a value of 0x000001A0. It indicates that the terminal topology manager detected that for the configured timeouts some device specific operations did not complete.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

TTM_WATCHDOG_TIMEOUT Parameters

PARAMETER	DESCRIPTION
1	Failure type - values listed below.
2	Pointer to the device.
3	Pointer to the worker thread.
4	Pointer to the callout routine.

Failure type

0x1 : A device assignment to a terminal is not making progress.

0x2 : Device's close callback is not making progress.

0x3 : Device's set-input-mode callback is not making progress.

0x4 : Device's set-display-state callback is not making progress.

0x5 : Setting device's built-in panel state is not making progress.

0x6 : Updating device's primary display visible state is not making progress.

Cause

The terminal topology manager detected that for the configured timeouts some device specific operations did not complete.

See Also

Bug Check Code Reference

Bug Check Bug Check 0x1A3: CALL_HAS_NOT_RETURNED_WATCHDOG_TIMEOUT_LIVEDUMP

7/2/2019 • 2 minutes to read • Edit Online

IMPORTANT

This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

A call has not returned within the timeout period.

The CALL_HAS_NOT_RETURNED_WATCHDOG_TIMEOUT_LIVEDUMP live dump has a value of 0x000001A3.

CALL_HAS_NOT_RETURNED_WATCHDOG_TIMEOUT_LIVEDUMP Parameters

The following parameters are displayed on the blue screen.

PARAMETER	DESCRIPTION
1	Process of a thread whose call has not returned promptly.
2	Thread whose call has not returned promptly.
3	Timeout in milliseconds.
4	dt nt!_PO_CALL_HAS_NOT_RETURNED_WATCHDOG

Bug Check Bug Check 0x1A4: DRIPS_SW_HW_DIVERGENCE_LIVEDUMP

7/2/2019 • 2 minutes to read • Edit Online

The DRIPS_SW_HW_DIVERGENCE_LIVEDUMP live dump has a value of 0x000001A4.

Software and hardware DRIPS divergence exceeds default/programmed threshold time.

(This code can never be used for a real bugcheck; it is used to identify live dumps.)

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

DRIPS_SW_HW_DIVERGENCE_LIVEDUMP Parameters

The following parameters are displayed on the blue screen.

PARAMETER	DESCRIPTION
1	Time spent in software DRIPS in microseconds.
2	Time spent in hardware DRIPS in microseconds.
3	Reserved.
4	Reserved.

Bug Check 0x1A5: USB_DRIPS_BLOCKER_SURPRISE_REMOVAL_LIVEDUMP

7/2/2019 • 2 minutes to read • Edit Online

The USB_DRIPS_BLOCKER_SURPRISE_REMOVAL_LIVEDUMP bug check has a value of 0x000001A5. It indicates that a USB device will be surprise removed because it is blocking DRIPS.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

USB_DRIPS_BLOCKER_SURPRISE_REMOVAL_LIVEDUMP Parameters

PARAMETER	DESCRIPTION
1	PDO for the blocking device.
2	Reserved.
3	Reserved.
4	Reserved.

Cause

A USB device is blocking the top level controller from powering down during modern standby and will be surprise removed as a result.

(This code can never be used for a real bugcheck; it is used to identify live dumps.)

See Also

Bug Check Code Reference

Bug Check 0x1A6: BLUETOOTH_ERROR_RECOVERY_LIVEDUMP

7/2/2019 • 2 minutes to read • Edit Online

The BLUETOOTH_ERROR_RECOVERY_LIVEDUMP bug check has a value of 0x000001A6. It indicates that the Bluetooth radio driver (bthport.sys) has initiated error recovery to attempt to recover and reset the radio from an irremediable internal condition.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

BLUETOOTH_ERROR_RECOVERY_LIVEDUMP Parameters

PARAMETER	DESCRIPTION
1	PDO for the Bluetooth radio (device).
2	Reserved.
3	Reserved.
4	Reserved.

Cause

The Bluetooth radio driver (bthport.sys) has initiated error recovery to attempt to recover and reset the radio from an irremediable internal condition.

(This code can never be used for a real bugcheck; it is used to identify live dumps.)

See Also

Bug Check Code Reference

Bug Check 0x1A7: SMB_REDIRECTOR_LIVEDUMP

7/13/2019 • 2 minutes to read • Edit Online

The SMB_REDIRECTOR_LIVEDUMP bug check has a value of 0x000001A7. It indicates that the SMB redirector has detected a problem and has captured a kernel dump to collect debug information.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

SMB_REDIRECTOR_LIVEDUMP Parameters

PARAMETER	DESCRIPTION
1	Reason Code - see values below.
2	See values below.
3	Reserved.
4	Reserved.

The SMB redirector has detected a problem and has captured a kernel dump to collect debug information.

Reason Code

```
0x1 : An I/O failed to complete in a reasonable amount of time.
    2 - Pointer to the connection object.
    3 - Reserved.
    4 - Reserved.
```

Cause

The SMB redirector has detected a problem and has captured a kernel dump to collect debug information.

A live dump with this bugcheck code will be generated only if the following registry value is set.

```
HKLM\System\CurrentControlSet\Services\Lanmanworkstation\Parameters [DWORD] LiveDumpFilter = 1
```

When this registry key is set and the RDR times out on IO, a livedump will occur.

(This code can never be used for a real bugcheck; it is used to identify live dumps.)

See Also

Bug Check Code Reference

Bug Check 0x1A8: VIDEO_DXGKRNL_BLACK_SCREEN_LIVEDUMP

7/2/2019 • 2 minutes to read • Edit Online

The VIDEO_DXGKRNL_BLACK_SCREEN_LIVEDUMP bug check has a value of 0x000001A8. It indicates that a user initiated DXGKRNL live dump for black screen scenarios has occurred.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

VIDEO_DXGKRNL_BLACK_SCREEN_LIVEDUMP Parameters

PARAMETER	DESCRIPTION
1	Source which triggered the DXGKRNL black screen live dump - listed below.
2	Reserved.
3	Reserved.
4	Reserved.

Source Values

0x1: Black screen hotkey generated DXGKRNL black screen live dump

0x2: Volume combo key generated DXGKRNL black screen live dump

0x4: Internal generated DXGKRNL black screen live dump

0x8: Long Power Button Hold (LPBH) generated DXGKRNL black screen live dump

Cause

User initiated DXGKRNL live dump for black screen scenarios. See the values for parameter 1 for the source of the triggered live dump.

(This code can never be used for a real bugcheck; it is used to identify live dumps.)

See Also

Bug Check Code Reference

Bug Check 0x1B0: VIDEO_MINIPORT_FAILED_LIVEDUMP

7/2/2019 • 2 minutes to read • Edit Online

The VIDEO_MINIPORT_FAILED_LIVEDUMP bug check has a value of 0x000001B0.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

VIDEO_MINIPORT_FAILED_LIVEDUMP Parameters

PARAMETER	DESCRIPTION
1	Reason Code. VALUES: 0x1 : *Add device failed.* 0x2 : *Start device failed.*
2	NTSTATUS
3	Reserved.
4	Reserved.

Cause

The DXGKRNL detected a problem and has captured a live dump to collect debug information. These livedumps are triggered by dxgkrnl when a video miniport driver failed.

(This code can never be used for a real bugcheck; it is used to identify live dumps.)

See Also

Bug Check Code Reference

Bug Check 0x1C4: DRIVER_VERIFIER_DETECTED_VIOLATION_LIVEDUMP

7/2/2019 • 15 minutes to read • Edit Online

The DRIVER_VERIFIER_DETECTED_VIOLATION_LIVEDUMP bug check has a value of 0x000001C4. This indicates that a device driver attempting to corrupt the system has been detected. This is because the driver was specified in the registry as being suspect (by the administrator) and the kernel has enabled substantial checking of this driver. For more information, see Driver Verifier.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

DRIVER_VERIFIER_DETECTED_VIOLATION_LIVEDUMP Parameters

PARAMETER	DESCRIPTION
1	The subclass of driver violation. See Values below.
2	See Values below.
3	See Values below.
4	See Values below.

Values

```
0x00081001: ID of the 'KsDeviceMutex' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Address of internal rule state (second argument to !ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to !ruleinfo).

0x00081002: ID of the 'KsStreamPointerClone' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Address of internal rule state (second argument to !ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to !ruleinfo).

0x00081003: ID of the 'KsStreamPointerLock' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x00081004: ID of the 'KsStreamPointerUnlock' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Address of internal rule state (second argument to !ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to !ruleinfo).

0x00081005: ID of the 'KsCallbackReturn' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x00081006: ID of the 'KsIrqlDeviceCallbacks' rule that was violated
```

```
0x00081006: ID of the 'KsIrqlDeviceCallbacks' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Address of internal rule state (second argument to !ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to !ruleinfo).

0x00081007: ID of the 'KsIrqlFilterCallbacks' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Address of internal rule state (second argument to !ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to !ruleinfo).

0x00081008: ID of the 'KsIrqlPinCallbacks' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Address of internal rule state (second argument to !ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to !ruleinfo).

0x00081009: ID of the 'KsIrqlDDIs' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x0008100A: ID of the 'KsFilterMutex' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Address of internal rule state (second argument to !ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to !ruleinfo).

0x0008100B: ID of the 'KsProcessingMutex' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Address of internal rule state (second argument to !ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to !ruleinfo).

0x00082001: ID of the 'KsTimedPinSetDeviceState' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Address of internal rule state (second argument to !ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to !ruleinfo).

0x00082002: ID of the 'KsTimedDeviceCallbacks' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Address of internal rule state (second argument to !ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to !ruleinfo).

0x00082003: ID of the 'KsTimedFilterCallbacks' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Address of internal rule state (second argument to !ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to !ruleinfo).

0x00082004: ID of the 'KsTimedPinCallbacks' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Address of internal rule state (second argument to !ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to !ruleinfo).

0x00082005: ID of the 'KsTimedProcessingMutex' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Address of internal rule state (second argument to !ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to !ruleinfo).

0x00071001: ID of the 'PcIrqlDDIs' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x00071003: ID of the 'PcIrqlIport' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x00071004: ID of the 'PcUnmapAllocatedPages' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Address of internal rule state (second argument to !ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to !ruleinfo).
```

```
0x00071005: ID of the 'PcAllocatedPages' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Address of internal rule state (second argument to !ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to !ruleinfo).

0x00071006: ID of the 'PcRegisterAdapterPower' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Address of internal rule state (second argument to !ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to !ruleinfo).

0x00071007: ID of the 'PcAddAdapterDevice' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x00071008: ID of the 'PcPropertyRequest' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x00071009: ID of the 'PcAllocateAndMapPages' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x0007100A: ID of the 'PcPoRequestPowerIrp' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x00072001: ID of the 'PcTimedWaveRtStreamSetState' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Address of internal rule state (second argument to !ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to !ruleinfo).

0x00020002: ID of the 'IrqlApcLte' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x00020003: ID of the 'IrqlDispatch' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x00020004: ID of the 'IrqlExAllocatePool' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x00020005: ID of the 'IrqlExApcLte1' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x00020006: ID of the 'IrqlExApcLte2' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x00020007: ID of the 'IrqlExApcLte3' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x00020008: ID of the 'IrqlExPassive' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Reserved
```

```
    Parameter 4 - Reserved

0x00020009: ID of the 'IrqlIoApcLte' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x0002000A: ID of the 'IrqlIoPassive1' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x0002000B: ID of the 'IrqlIoPassive2' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x0002000C: ID of the 'IrqlIoPassive3' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x0002000D: ID of the 'IrqlIoPassive4' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x0002000E: ID of the 'IrqlIoPassive5' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x0002000F: ID of the 'IrqlKeApcLte1' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x00020010: ID of the 'IrqlKeApcLte2' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x00020011: ID of the 'IrqlKeDispatchLte' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x00020015: ID of the 'IrqlKeReleaseSpinLock' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x00020016: ID of the 'IrqlKeSetEvent' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x00020019: ID of the 'IrqlMmApcLte' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x0002001A: ID of the 'IrqlMmDispatch' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x0002001B: ID of the 'IrqlObPassive' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
```

```
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x0002001C: ID of the 'IrqlPsPassive' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x0002001D: ID of the 'IrqlReturn' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Address of internal rule state (second argument to !ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to !ruleinfo).

0x0002001E: ID of the 'IrqlRtlPassive' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x0002001F: ID of the 'IrqlZwPassive' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x00020022: ID of the 'IrqlIoDispatch' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x00040003: ID of the 'CriticalRegions' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Address of internal rule state (second argument to !ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to !ruleinfo).

0x00040006: ID of the 'QueuedSpinLock' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Address of internal rule state (second argument to !ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to !ruleinfo).

0x00040007: ID of the 'QueuedSpinLockRelease' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Address of internal rule state (second argument to !ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to !ruleinfo).

0x00040009: ID of the 'SpinLock' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Address of internal rule state (second argument to !ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to !ruleinfo).

0x0004000A: ID of the 'SpinlockRelease' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Address of internal rule state (second argument to !ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to !ruleinfo).

0x0004000E: ID of the 'GuardedRegions' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Address of internal rule state (second argument to !ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to !ruleinfo).

0x0004100B: ID of the 'RequestedPowerIrp' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x0004100F: ID of the 'IoSetCompletionExCompleteIrp' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Address of internal rule state (second argument to !ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to !ruleinfo).

0x00043006: ID of the 'PnpRemove' rule that was violated.
```

```
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x00091001: ID of the 'NdisOidComplete' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Address of internal rule state (second argument to !ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to !ruleinfo).

0x00091002: ID of the 'NdisOidDoubleComplete' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Address of internal rule state (second argument to !ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to !ruleinfo).

0x0009100E: ID of the 'NdisOidDoubleRequest' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Address of internal rule state (second argument to !ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to !ruleinfo).

0x00092003: ID of the 'NdisTimedOidComplete' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Address of internal rule state (second argument to !ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to !ruleinfo).

0x0009200D: ID of the 'NdisTimedDataSend' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Address of internal rule state (second argument to !ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to !ruleinfo).

0x0009200F: ID of the 'NdisTimedDataHang' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Address of internal rule state (second argument to !ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to !ruleinfo).

0x00092010: ID of the 'NdisFilterTimedPauseComplete' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Address of internal rule state (second argument to !ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to !ruleinfo).

0x00092011: ID of the 'NdisFilterTimedDataSend' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Address of internal rule state (second argument to !ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to !ruleinfo).

0x00092012: ID of the 'NdisFilterTimedDataReceive' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Address of internal rule state (second argument to !ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to !ruleinfo).

0x00093004: ID of the 'WlanAssociation' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Address of internal rule state (second argument to !ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to !ruleinfo).

0x00093005: ID of the 'WlanConnectionRoaming' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Address of internal rule state (second argument to !ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to !ruleinfo).

0x00093006: ID of the 'WlanDisassociation' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Address of internal rule state (second argument to !ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to !ruleinfo).

0x00093101: ID of the 'WlanAssert' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved
```

```
0x00094007: ID of the 'WlanTimedAssociation' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Address of internal rule state (second argument to !ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to !ruleinfo).

0x00094008: ID of the 'WlanTimedConnectionRoaming' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Address of internal rule state (second argument to !ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to !ruleinfo).

0x00094009: ID of the 'WlanTimedConnectRequest' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Address of internal rule state (second argument to !ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to !ruleinfo).

0x0009400B: ID of the 'WlanTimedLinkQuality' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Address of internal rule state (second argument to !ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to !ruleinfo).

0x0009400C: ID of the 'WlanTimedScan' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule condition.
    Parameter 3 - Address of internal rule state (second argument to !ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to !ruleinfo).
```

Bug Check 0x1C5: IO_THREADPOOL_DEADLOCK_LIVEDUMP

7/2/2019 • 2 minutes to read • Edit Online

The IO_THREADPOOL_DEADLOCK_LIVEDUMP bug check has a value of 0x000001C5. This indicates a kernel mode threadpool encountered a deadlock situation.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

IO_THREADPOOL_DEADLOCK_LIVEDUMP Parameters

PARAMETER	DESCRIPTION
1	Pool Number. 0x0 : ExPoolUntrusted
2	Pointer to the PEX_WORK_QUEUE
3	Reserved
4	Reserved

Bug Check 0x1C6: FAST_ERESOURCE_PRECONDITION_VIOLATION

7/13/2019 • 3 minutes to read • Edit Online

The FAST_ERESOURCE_PRECONDITION_VIOLATION bug check has a value of 0x000001C6. It indicates that a current thread is performing an invalid call to a fast resource routine.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

FAST_ERESOURCE_PRECONDITION_VIOLATION Parameters

PARAMETER	DESCRIPTION
1	Violation type. See values below.
2	See values below.
3	See values below.
4	See values below.

Violation type

```
0x0 : The Irql of the caller was greater than the maximum allowed
      Irql for the routine.
    2 - Irql of the caller.
    3 - Maximum allowed Irql of the routine.
0x1 : The caller specified an invalid (i.e. uninitialized) owner
      entry.
    2 - Pointer to the owner entry.
0x2 : The caller specified an owner entry that was already
      associated with a lock acquisition.
    2 - Pointer to the owner entry.
    3 - Pointer to the resource to which the owner entry is already
        associated.
0x3 : The caller passed a legacy resource to a fast resource
      routine.
    2 - Pointer to the resource.
0x4 : The caller specified a resource that has outstanding lock
      acquisitions.
    2 - Pointer to the resource.
0x5 : The caller was executing inside of a DPC.
0x6 : The caller was executing inside of a Special Kernel APC.
0x7 : The caller did not ensure Normal Kernel APCs were disabled.
0x8 : The caller specified an owner entry that was not associated
      with a lock acquisition of the specified resource.
    2 - Pointer to the resource specified.
    3 - Pointer to the owner entry.
    4 - Pointer to the resource with which the owner entry is
        associated.
0x9 : The caller specified an owner entry that was not associated
```

```
      with the calling thread.
    2 - Pointer to the owner entry.
    3 - Pointer to the thread with which the owner entry is
      associated.
0xa : The caller specified an owner entry which has been disowned.
    2 - Pointer to the owner entry.
0xb : The caller specified an owner entry with a different
      acquisition type than the caller indicated.
    2 - Pointer to the owner entry.
    3 - High-16 bits indicate the acquisition type of the owner
        entry. Bottom-16 bits indicate the acquisition type
        specified. 1 = Shared, 0 = Exclusive.
0xc : The caller specified an owner entry that was not associated
      with a lock acquisition of the specified resource.
    2 - Pointer to the resource specified.
    3 - Pointer to the owner entry.
    4 - Pointer to the resource with which the owner entry is
        associated.
0xd : The caller specified an owner entry that has not been
      disowned.
    2 - Pointer to the owner entry.
0xe : The caller passed a fast resource to a legacy routine that
      does not support fast resources.
    2 - Pointer to the resource.
0xf : The caller passed a fast resource to a legacy routine that
      supports fast resources, but the fast resource was not
      initialized with EX_FAST_RESOURCE_ENABLE_LEGACY_APIS.
    2 - Pointer to the resource.
0x10 : The caller passed invalid flags to ExInitializeFastResource.
    2 - Pointer to the resource specified.
    3 - The flags specified.
0x11 : The caller passed a thread other than the current thread to
       ExReleaseResourceForThreadLite.
    2 - Pointer to the resource specified.
    3 - The thread specified.
0x12 : The caller attempted to disown a resource that had been
       recursively acquired exclusive by the current thread.
    2 - Pointer to the resource specified.
0x13 : The caller attempted to convert a resource acquisition while
       the calling thread had outstanding recursive acquisitions of
       the resource.
    2 - Pointer to the resource specified.
0x14 : A thread exited with outstanding lock acquisitions.
    2 - The thread.
    3 - Pointer to one of the outstanding owner entries.
0x15 : A thread exited with outstanding disowned lock acquisitions.
    2 - The thread.
    3 - Pointer to one of the outstanding owner entries.
0x16 : A call to ExConvertExclusiveToSharedLite was made by a thread
       that did not hold the specified resource exclusive.
    2 - Pointer to the resource.
    3 - Pointer to the thread.
```

Cause

The current thread is performing an invalid call to a fast resource routine.

Resolution

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause.

See Also

Bug Check 0x1C7: STORE_DATA_STRUCTURE_CORRUPTION

7/13/2019 • 2 minutes to read • Edit Online

The STORE_DATA_STRUCTURE_CORRUPTION bug check has a value of 0x000001C7. It indicates that the store component detected a corruption in its data structures.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

STORE_DATA_STRUCTURE_CORRUPTION Parameters

PARAMETER	DESCRIPTION
1	Corruption ID. See values below.
2	See values below.
3	See values below.
4	See values below.

Corruption ID

```
 0x0 : A chunk heap buffer's hash doesn't match.
    2 - Chunk heap buffer whose hash didn't match.
    3 - Expected buffer hash.
    4 - Page frame number of the corrupted page.

 0x1 : An unhandled exception occurred on the store thread and a chunk heap buffer's hash doesn't match, which
is likely the source of the exception.
    2 - Chunk heap buffer whose hash didn't match.
    3 - Expected buffer hash.
    4 - Page frame number of the corrupted page.

 0x2 : Page data appears corrupt during a read and the corresponding page record's heap buffer hash doesn't
match.
    2 - Chunk heap buffer whose hash didn't match containing the page record of the data being read.
    3 - Expected buffer hash.
    4 - Page frame number of the corrupted page.

 0x3 : Page data appears corrupt during a read and the corresponding page record has changed since the start
of the read operation.
    2 - Pointer to the page location information snapped from the page record that was found when the read was
initiated.
    3 - Pointer to the page record currently in the page tree for the same page key.
    4 - Reserved.
```

Cause

The store component detected a corruption in its data structures.

This bugcheck can occur by memory corruption due to physical memory access. The causes for physical memory corruption include:

1. Defective RAM hardware
2. A driver or device incorrectly modifying physical pages via an incorrect DMA operation or associated MDL.
3. Corruption caused by a hardware device or firmware corrupting memory, such as firmware illegally modifying physical pages across a power transition.

For more information on Windows memory manager, see Windows Internals 7th Edition Part 1 by Pavel Yosifovich, Mark E. Russinovich, David A. Solomon and Alex Ionescu.

Resolution

Windows Memory Diagnostics Tool

To investigate if this bug check is caused by defective RAM hardware, run the Windows Memory Diagnostics tool. In the control panel search box, type Memory, and then click *Diagnose your computer's memory problems*. After the test is run, use Event viewer to view the results under the System log. Look for the *MemoryDiagnostics-Results* entry to view the results.

See Also

Bug Check Code Reference

Windows Kernel-Mode Memory Manager

Bug Check 0x1C8: MANUALLY_INITIATED_POWER_BUTTON_HOLD

7/2/2019 • 2 minutes to read • Edit Online

The MANUALLY_INITIATED_POWER_BUTTON_HOLD has a value of 0x000001C8.

The system was configured to initiate a bugcheck when the user holds the power button for a specified length of time. This is a diagnostic bugcheck used to capture a dump when the system is about to be hard reset with a long power button hold.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Note that when this bug check occurs instead of the standard "blue screen" being displayed, a black background with the following text is displayed along with a % completion indicator:

"Please release the power button. We just need a few more seconds to shut down."

MANUALLY_INITIATED_POWER_BUTTON_HOLD Parameters

The following parameters are displayed on the blue screen.

PARAMETER	DESCRIPTION
1	Time in milliseconds the power button was held down.
2	Pointer to nt!_POP_POWER_BUTTON_TRIAGE_BLOCK.
3	Reserved.
4	Reserved.

Bug Check 0x1C9: USER_MODE_HEALTH_MONITOR_LIVEDUMP

7/2/2019 • 2 minutes to read • Edit Online

The USER_MODE_HEALTH_MONITOR_LIVEDUMP bug check has a value of 0x000001C9. It indicates that one or more critical user mode components failed to satisfy a health check.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

USER_MODE_HEALTH_MONITOR_LIVEDUMP Parameters

PARAMETER	DESCRIPTION
1	Process that failed to satisfy a health check within the configured timeout.
2	Health monitoring timeout (seconds).
3	Watchdog source. In combination with process address this helps to identify the source. See below for possible values. These values are shared with USER_MODE_HEALTH_MONITOR.
4	Reserved.

Watchdog Source Values

```
  0  : WatchdogSourceDefault
       Source was not specified
  1  : WatchdogSourceRhsCleanup
       Monitors that RHS process goes away when
       terminating on graceful exit
  2  : WatchdogSourceRhsResourceDeadlockBugcheckNow
       RHS was asked to immediately bugcheck machine
       on resource deadlock
  3  : WatchdogSourceRhsExceptionFromResource
       Resource has leaked unhandled exception from an entry point,
       RHS is terminating and this watchdog monitors that
       process will go away
  4  : WatchdogSourceRhsUnhandledException
       Unhandled exception in RHS.
       RHS is terminating and this watchdog monitors that
       process will go away
  5  : WatchdogSourceRhsResourceDeadlock
       Monitors that RHS process goes away when
       terminating on resource deadlock
  6  : WatchdogSourceRhsResourceTypeDeadlock
       Monitors that RHS process goes away when
       terminating on resource type deadlock
  7  : WatchdogSourceClussvcUnhandledException
       Unhandled exception in clussvc.
       clussvc is terminating and this watchdog monitors that
       process will go away
  8  : WatchdogSourceClussvcBugcheckMessageRecieved
       Another cluster node has sent message asking to bugcheck this node.
  9  : WatchdogSourceClussvcWatchdogBugcheck
       User mode watchdog has expired and created netft watchdog
       to bugchecked the node.
 0xA : WatchdogSourceClussvcIsAlive
       Cluster service sends heartbeat to netft every 500 millseconds.
       By default, netft expects at least 1 heartbeat per second.
       If this watchdog was triggered that means clussvc is not getting
       CPU to send heartbeats.
0x65 : WatchdogSourceRhsResourceDeadlockPhysicalDisk
        A subclass of WatchdogSourceRhsResourceDeadlock.
0x66 : WatchdogSourceRhsResourceDeadlockStoragePool
        A subclass of WatchdogSourceRhsResourceDeadlock.
0x67 : WatchdogSourceRhsResourceDeadlockFileServer
        A subclass of WatchdogSourceRhsResourceDeadlock.
0x68 : WatchdogSourceRhsResourceDeadlockSODAFileServer
        A subclass of WatchdogSourceRhsResourceDeadlock.
0x69 : WatchdogSourceRhsResourceDeadlockStorageReplica
        A subclass of WatchdogSourceRhsResourceDeadlock.
0x6A : WatchdogSourceRhsResourceDeadlockStorageQOS
        A subclass of WatchdogSourceRhsResourceDeadlock.
0x6B : WatchdogSourceRhsResourceDeadlockStorageNFSV2
        A subclass of WatchdogSourceRhsResourceDeadlock.
0xC9 : WatchdogSourceRhsResourceTypeDeadlockPhysicalDisk
        A subclass of WatchdogSourceRhsResourceTypeDeadlock.
0xCA : WatchdogSourceRhsResourceTypeDeadlockStoragePool
        A subclass of WatchdogSourceRhsResourceTypeDeadlock.
0xCB : WatchdogSourceRhsResourceTypeDeadlockFileServer
        A subclass of WatchdogSourceRhsResourceTypeDeadlock.
0xCC : WatchdogSourceRhsResourceTypeDeadlockSODAFileServer
        A subclass of WatchdogSourceRhsResourceTypeDeadlock.
0xCD : WatchdogSourceRhsResourceTypeDeadlockStorageReplica
        A subclass of WatchdogSourceRhsResourceTypeDeadlock.
0xCE : WatchdogSourceRhsResourceTypeDeadlockStorageQOS
        A subclass of WatchdogSourceRhsResourceTypeDeadlock.
0xCF : WatchdogSourceRhsResourceTypeDeadlockStorageNFSV2
        A subclass of WatchdogSourceRhsResourceTypeDeadlock.
```

Cause

One or more critical user mode components failed to satisfy a health check.

Hardware mechanisms such as watchdog timers can detect that basic kernel services are not executing. However, resource starvation issues, including memory leaks, lock contention, and scheduling priority misconfiguration, may block critical user mode components without blocking DPCs or draining the nonpaged pool.

Kernel components can extend watchdog timer functionality to user mode by periodically monitoring critical applications. This livedump indicates that a user mode health check failed in a manner such that we will attempt to terminate this application and will keep monitoring if termination completes in time. If termination does not complete in time, then the machine will be bugchecked It restores critical services by rebooting and/or allowing application failover to other servers.

(This code can never be used for a real bugcheck; it is used to identify live dumps.)

See Also

Bug Check Code Reference

Bug Check 0x1CA: SYNTHETIC_WATCHDOG_TIMEOUT

7/2/2019 • 2 minutes to read • Edit Online

The SYNTHETIC_WATCHDOG_TIMEOUT bug check has a value of 0x000001CA. A system wide watchdog has expired. This indicates that the system is hung and not processing timer ticks.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

SYNTHETIC_WATCHDOG_TIMEOUT Parameters

PARAMETER	DESCRIPTION
1	The time since the watchdog was last reset, in interrupt time.
2	The current interrupt time.
3	The current QPC timestamp.
4	The index of the clock processor.

Cause

A system wide watchdog has expired. This indicates that the system is hung and not processing timer ticks.

See Also

Bug Check Code Reference

Bug Check 0x1CB: INVALID_SILO_DETACH

7/2/2019 • 2 minutes to read • Edit Online

The INVALID_SILO_DETACH bug check has a value of 0x000001CB. It indicates that a thread failed to detach from a silo before exiting.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

INVALID_SILO_DETACH Parameters

PARAMETER	DESCRIPTION
1	Pointer to the attached thread.
2	Previously attached silo.
3	Pointer to the thread's process.
4	Reserved.

Cause

A thread failed to detach from a silo before exiting.

See Also

Bug Check Code Reference

Bug Check Bug Check 0x1CC: EXRESOURCE_TIMEOUT_LIVEDUMP

7/2/2019 • 2 minutes to read • Edit Online

IMPORTANT

This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

The EXRESOURCE_TIMEOUT_LIVEDUMP bug check has a value of 0x000001CC.

A kernel ERESOURCE has timed out. This can indicate a deadlock condition or heavy contention which can cause performance issues.

EXRESOURCE_TIMEOUT_LIVEDUMP Parameters

The following parameters are displayed on the blue screen.

PARAMETER	DESCRIPTION
1	The ERESOURCE that has timed out.
2	The thread that detected the timeout
3	The ERESOURCE contention count
4	The configured timeout in seconds

See Also

Bug Check Code Reference

Bug Check Bug Check 0x1CD: INVALID_CALLBACK_STACK_ADDRESS

7/2/2019 • 2 minutes to read • Edit Online

IMPORTANT

This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

The INVALID_CALLBACK_STACK_ADDRESS bug check has a value of 0x000001CD. The callback stack is a user mode address which is illegal.

INVALID_CALLBACK_STACK_ADDRESS Parameters

None

Bug Check Bug Check 0x1CE: INVALID_KERNEL_STACK_ADDRESS

7/2/2019 • 2 minutes to read • Edit Online

IMPORTANT

This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

The INVALID_KERNEL_STACK_ADDRESS bug check has a value of 0x000001CE. An invalid initial kernel stack address was encountered during the context switch.

INVALID_KERNEL_STACK_ADDRESS Parameters

None

Bug Check Bug Check 0x1CF: HARDWARE_WATCHDOG_TIMEOUT

7/2/2019 • 2 minutes to read • Edit Online

IMPORTANT

This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

The HARDWARE_WATCHDOG_TIMEOUT bug check has a value of 0x000001CF. This indicates that the system is hung and not processing timer ticks.

HARDWARE_WATCHDOG_TIMEOUT Parameters

PARAMETER	DESCRIPTION
1	The time since the watchdog was last reset, in interrupt time.
2	The current interrupt time.
3	The current QPC timestamp.
4	The index of the clock processor.

Bug Check Bug Check 0x1D0: ACPI_FIRMWARE_WATCHDOG_TIMEOUT

7/2/2019 • 2 minutes to read • Edit Online

IMPORTANT

This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

The ACPI_FIRMWARE_WATCHDOG_TIMEOUT bug check has a value of 0x000001D0.

ACPI driver failed to complete an operation in expected alloted time.

ACPI_FIRMWARE_WATCHDOG_TIMEOUT Parameters

The following parameters are displayed on the blue screen.

PARAMETER	DESCRIPTION
1	Pointer to AMLI Context
2	Pointer to Unicode Name of the Aml Context
3	Pointer to ACPI Device Extension.
4	Pointer to ACPI Triage Block.

Bug Check Bug Check 0x1D1: TELEMETRY_ASSERTS_LIVEDUMP

7/2/2019 • 2 minutes to read • Edit Online

IMPORTANT

This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

The TELEMETRY_ASSERTS_LIVEDUMP bug check has a value of 0x000001D1.

A Telemetry Assert verification failed.

This code can never be used for a real bugcheck; it is used to identify live dumps including device telemetry.

To troubleshoot this issue, inspect the callstack to see why the expression in MICROSOFT_TELEMETRY_ASSERT(expression) is invalid.

TELEMETRY_ASSERTS_LIVEDUMP Parameters

PARAMETER	DESCRIPTION
1	Rva
2	ModuleName
3	TimeStamp
4	SizeOfImage

Bug Check Bug Check 0x1D2: WORKER_THREAD_INVALID_STATE

7/2/2019 • 2 minutes to read • Edit Online

IMPORTANT

This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

The WORKER_THREAD_INVALID_STATE bug check has a value of 0x000001D2.

This error indicates that an executive worker thread is in an invalid state.

WORKER_THREAD_INVALID_STATE Parameters

PARAMETER	DESCRIPTION
1	Type of failure
2	Address of the worker thread
3	Reserved
4	Reserved

Parameter 1 Values

0x0 : A worker thread in the process of terminating has outstanding I/O

2 - Address of the worker thread

3 - Reserved

4 - Reserved

Bug Check Bug Check 0x1D3: WFP_INVALID_OPERATION

7/13/2019 • 2 minutes to read • Edit Online

IMPORTANT

This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

The WFP_INVALID_OPERATION bug check has a value of 0x000001D3. This indicates that a Windows Filtering Platform callout performed an invalid operation.

WFP_INVALID_OPERATION Parameters

PARAMETER	DESCRIPTION
1	The subtype of the bugcheck.
2	Reserved
3	Reserved
4	Reserved

Parameter 1 Values

0x1 : Callout injected an NBL with multiple NET_BUFFERS inbound.

2 - Reserved.

3 - Pointer to NBL.

4 - Reserved.

Resolution

The **!analyze** debug extension displays information about the bug check and can be very helpful in determining the root cause.

Bug Check Bug Check 0x1D4: UCMUCSI_LIVEDUMP

7/2/2019 • 2 minutes to read • Edit Online

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

The UCMUCSI_LIVEDUMP live dump has a value of 0x000001D4.

The UcmUcsi.sys driver has encountered an error. UcmUcsi.sys is an-in box USB Connector Manager UCSI client driver. For more information, see USB Type-C Connector System Software Interface (UCSI) driver.

UCMUCSI_LIVEDUMP Parameters

PARAMETER	DESCRIPTION
1	Type of failure - see values below
2	The UCSI command value.
3	If non-zero, the pointer to additional information (dt UcmUcsiCx!UCMUCSICX_TRIAGE).
4	Reserved.

Type of Failure

0x0 : A UCSI command has timed out because the firmware did not respond to the command in time.

0x1 : A UCSI command execution failed either because the client driver returned failure or because the firmware returned an error code.

Bug Check 0x1D5: DRIVER_PNP_WATCHDOG

10/25/2019 • 2 minutes to read • Edit Online

The DRIVER_PNP_WATCHDOG bug check has a value of 0x000001D5. This indicates that a driver has failed to complete a PnP operation within a specific time.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

DRIVER_PNP_WATCHDOG Parameters

PARAMETER	DESCRIPTION
1	First few character of the service associated with the devnode.
2	Pointer to the nt!TRIAGE_PNP_WATCHDOG on Win10 RS4 and higher.
3	Thread responsible for the PnP Watchdog.
4	Milliseconds elapsed since the watchdog was armed.

Cause

This indicates that a driver has failed to complete a PnP operation within a specific time. The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause.

See Also

Bug Check Code Reference

Bug Check 0x1D6: WORKER_THREAD_RETURNED_WITH_NON_DEFAULT_WORKLOAD_CLASS

7/2/2019 • 2 minutes to read • Edit Online

The WORKER_THREAD_RETURNED_WITH_NON_DEFAULT_WORKLOAD_CLASS bug check has a value of 0x000001D6. It indicates that a worker thread changed its workload class and did not revert it before returning.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

WORKER_THREAD_RETURNED_WITH_NON_DEFAULT_WORKLOAD_CLASS Parameters

PARAMETER	DESCRIPTION
1	Address of worker routine (use ln on this to find the responsible driver)
2	Current workload class value.
3	WorkItem parameter.
4	WorkItem address.

Cause

A worker thread changed its workload class and did not revert it before returning.

See Also

Bug Check Code Reference

Bug Check 0x1D7: EFS_FATAL_ERROR

7/2/2019 • 2 minutes to read • Edit Online

The EFS_FATAL_ERROR bug check has a value of 0x000001D7. It indicates that an EFS error condition has occured such that cannot be handled without data loss or data corruption.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

EFS_FATAL_ERROR Parameters

PARAMETER	DESCRIPTION
1	Bug Check Subclass: **01** - Pre-offloading failure.
2	NTSTATUS return code of the operation.
3	The current IRP at the time of failure.
4	File encryption context at the time of failure.

Cause

An EFS error condition has occured such that cannot be handled without data loss or data corruption.

Resolution

The !analyze debug extension displays information about the bug check and can be helpful in determining the root cause.

Examine parameter 2 the NTSTATUS field to try and determine why NT_SUCCESS was not returned. This is the expected and only allowed value for file systems that call crypto pre-offloading.

Use the debugger !IRP command to investigate parameter 3 for a possible conflicting IRP code or other issues.

See Also

Bug Check Code Reference

Bug Check 0x1D8: UCMUCSI_FAILURE

7/2/2019 • 2 minutes to read • Edit Online

The UCMUCSI_FAILURE bug check has a value of 0x000001D8. It indicates that the UCSI class extension has encountered an error.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

UCMUCSI_FAILURE Parameters

PARAMETER	DESCRIPTION
1	Type of failure. VALUES: **0x0** : A UCSI command has timed out because the firmware did not respond to the command in time. **0x1** : A UCSI command execution failed either because the client driver returned failure or because the firmware returned an error code.
2	The UCSI command value.
3	If non-zero, the pointer to additional information (use `dt UcmUcsiCx!UCMUCSICX_TRIAGE`).
4	Reserved.

Cause

The UcmUcsi driver has encountered an error. The driver has found settings to trigger a system crash instead of a livedump.

Resolution

A UCSI Command typically fails when UCSI firmware is not responsive and UcmUcsiCx times out on a UCSI command or the UCSI firmware has indicated an error in response to a critical UCSI command sent by UcmUcsiCx.

Run `!rcdrkd.rcdrlogdump UcmUcsiCx` for more information on the cause of this failure.

For more information on analyzing this bug check, see this blog post - Debugging UCSI firmware failures.

See Also

Bug Check Code Reference

Bug Check 0x1D9: HAL_IOMMU_INTERNAL_ERROR

7/2/2019 • 2 minutes to read • Edit Online

The HAL_IOMMU_INTERNAL_ERROR bug check has a value of 0x000001D9. It indicates that an internal error was detected in the HAL IOMMU library.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

HAL_IOMMU_INTERNAL_ERROR Parameters

PARAMETER	DESCRIPTION
1	Indicates the failed operation. See values below.
2	See values below.
3	See values below.
4	See values below.

Failed Operation Values

```
0x00 : Failed to delete IOMMU domain
     Parameter 2 - Status
     Parameter 3 - Pointer to the IOMMU domain object

0x01 : Failed to unmap pages from IOMMU domain
     Parameter 2 - Status
     Parameter 3 - Pointer to the IOMMU domain object
     Parameter 4 - Logical address

0x02 : Failed to leave IOMMU domain
     Parameter 2 - Status
     Parameter 3 - Pointer to the IOMMU domain object
```

Cause

An internal error was detected in the HAL IOMMU library.

See Also

Bug Check Code Reference

Bug Check 0x1DB: IPI_WATCHDOG_TIMEOUT

7/2/2019 • 2 minutes to read • Edit Online

The IPI_WATCHDOG_TIMEOUT bug check has a value of 0x000001DB. It indicates that a processor has been stuck in an IPI loop for more than the allowed time.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

IPI_WATCHDOG_TIMEOUT Parameters

PARAMETER	DESCRIPTION
1	Indicates QPC frequency.
2	Indicates the current QPC.
3	Indicates the baseline QPC.
4	Reserved.

Cause

A processor has been stuck in an IPI loop for more than the allowed time.

See Also

Bug Check Code Reference

Bug Check 0x1DC: DMA_COMMON_BUFFER_VECTOR_ERROR

10/25/2019 • 2 minutes to read • Edit Online

The DMA_COMMON_BUFFER_VECTOR_ERROR bug check has a value of 0x000001DC. It indicates that a driver has misused the DMA vectored common buffer APIs.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

DMA_COMMON_BUFFER_VECTOR_ERROR Parameters

PARAMETER	DESCRIPTION
1	Indicates the type of failure. See values below.
2	See values below.
3	See values below.
4	See values below.

Type of Failure

```
0x01 : Wrong IRQL
    2 - Current IRQL.

x02 : Vector not empty.
    2 - Index of remaining buffer.
    3 - Virtual Address of remaining buffer.
    4 - Logical address of remaining buffer.

0x03 : Index out of bounds.
    2 - Number of available entries.
    3 - Index requested.

0x04 : Index freed.
    2 - Index requested.

0x05 : Common buffer leaked.
```

Cause

A driver has misused the DMA vectored common buffer APIs. The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause.

See Also

Bug Check Bug Check 0x356: XBOX_ERACTRL_CS_TIMEOUT

7/2/2019 • 2 minutes to read • Edit Online

IMPORTANT

This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

The XBOX_ERACTRL_CS_TIMEOUT bug check has a value of 0x00000356.

The eractrl.sys driver was not able to transition the Xbox console to or from connected standby after an extended amount of time. The eractrl.sys driver is for a KM driver that bootstraps a supporting VM and helps manage the VM lifetime.

XBOX_ERACTRL_CS_TIMEOUT Parameters

PARAMETER	DESCRIPTION
1	CS exit
2	Reserved
3	Reserved
4	Reserved

Bug Check 0xBFE: BC_BLUETOOTH_VERIFIER_FAULT

10/25/2019 • 2 minutes to read • Edit Online

The BC_BLUETOOTH_VERIFIER_FAULT bug check has a value of 0x00000BFE. This indicates that a driver has caused a violation.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

BC_BLUETOOTH_VERIFIER_FAULT Parameters

PARAMETER	DESCRIPTION
1	The subtype of the Bluetooth verifier fault. `0x1 : An attempt was made to submit a Bluetooth Request Block that is already in use 2 - Brb pointer 3 - Reserved 4 - Reserved 0x2 : An attempt was made to free a Bluetooth Request Block that is in use 2 - Brb pointer 3 - Reserved 4 - Reserved 0x3 : An attempt was made to allocate or initialize an invalid BRB type 2 - Brb pointer 3 - pdo extension (if available) 4 - Reserved 0x4 : Invalid Bluetooth Request Block pointer was submitted 2 - Brb pointer 3 - Reserved 4 - Reserved 0x5 : A Bluetooth Request Block with an invalid size was submitted 2 - Brb pointer 3 - Actual Size 4 - Expected Size 0x6 : The IOCTL_BTH_GET_DEVICE_INFO was called with invalid parameters 2 - Reserved 3 - Reserved 4 - Reserved 0x7 : BRB_L2CA_UNREGISTER_SERVER was submitted with an invalid server handle 2 - Server handle 3 - Reserved 4 - Reserved 0x8 : BRB_L2CA_CLOSE_CHANNEL was submitted with an invalid channel handle 2 - Brb pointer 3 - Channel handle 4 - Reserved 0x9 : BRB_SCO_UNREGISTER_SERVER was submitted with an invalid server handle 2 - Server handle 3 - Reserved 4 - Reserved`
2	See parameter 1
3	See parameter 1
4	See parameter 1

Resolution

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause. Parameter 1 describes the type of violation. Look at the call stack to determine the misbehaving driver.

Bug Check 0xBFF: BC_BTHMINI_VERIFIER_FAULT

10/24/2019 • 2 minutes to read • Edit Online

The BC_BTHMINI_VERIFIER_FAULT bug check has a value of 0x00000BFF. This indicates that The Bluetooth miniport extensible driver verifier has caught a violation.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

BC_BTHMINI_VERIFIER_FAULT Parameters

PARAMETER	DESCRIPTION
1	The subtype of the Bluetooth verifier fault. `0x1 : An attempt was made to return a packet with type that mis-matched its original request. 2 - Returned packet type 3 - Expected packet type 4 - Reserved 0x2 : An attempt was made to return an unexpected status code and caused the packet to be discarded. 2 - Unexpected return status 3 - Reserved 4 - Reserved 0x3 : Incorrect output buffer size was returned to indicate number of bytes written by the lower transport driver. 2 - Unexpected buffer size 3 - Expected buffer size 4 - Reserved`
2	See parameter 1
3	See parameter 1
4	See parameter 1

Resolution

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause. Parameter 1 describes the type of violation. Look at the call stack to determine the misbehaving driver.

Bug Check 0x20001: HYPERVISOR_ERROR

7/2/2019 • 2 minutes to read • Edit Online

The HYPERVISOR_ERROR bug check has a value of 0x00020001. This indicates that the hypervisor has encountered a fatal error.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

HYPERVISOR_ERROR Parameters

PARAMETER	DESCRIPTION
1	Reserved
2	Reserved
3	Reserved
4	Reserved

Resolution

The **!analyze** debug extension displays information about the bug check and can be very in determining the root cause.

Bug Check 0x1000007E: SYSTEM_THREAD_EXCEPTION_NOT_HANDLED_M

7/2/2019 • 2 minutes to read • Edit Online

The SYSTEM_THREAD_EXCEPTION_NOT_HANDLED_M bug check has a value of 0x1000007E. This indicates that a system thread generated an exception which the error handler did not catch.

Bug check 0x1000007E has the same meaning and parameters as **bug check 0x7E** (SYSTEM_THREAD_EXCEPTION_NOT_HANDLED).

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Resolution

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause.

Bug Check 0x1000007F: UNEXPECTED_KERNEL_MODE_TRAP_M

10/25/2019 • 2 minutes to read • Edit Online

The UNEXPECTED_KERNEL_MODE_TRAP_M bug check has a value of 0x1000007F. This indicates that a trap was generated by the Intel CPU and the kernel failed to catch this trap.

Bug check 0x1000007F has the same meaning and parameters as **bug check 0x7F** (UNEXPECTED_KERNEL_MODE_TRAP).

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Resolution

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause.

Bug Check 0x1000008E: KERNEL_MODE_EXCEPTION_NOT_HANDLED_M

10/25/2019 • 2 minutes to read • Edit Online

The KERNEL_MODE_EXCEPTION_NOT_HANDLED_M bug check has a value of 0x1000008E. This indicates that a kernel-mode program generated an exception which the error handler did not catch.

Bug check 0x1000008E has the same meaning and parameters as **bug check 0x8E** (KERNEL_MODE_EXCEPTION_NOT_HANDLED).

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Resolution

The **!analyze** debug extension displays information about the bug check and can be helpful in determining the root cause.

Bug Check 0x100000EA: THREAD_STUCK_IN_DEVICE_DRIVER_M

7/2/2019 • 2 minutes to read • Edit Online

The THREAD_STUCK_IN_DEVICE_DRIVER_M bug check has a value of 0x100000EA. This indicates that a thread in a device driver is endlessly spinning.

Bug check 0x100000EA has the same meaning and parameters as **bug check 0xEA** (THREAD_STUCK_IN_DEVICE_DRIVER).

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

Bug Check 0x4000008A: THREAD_TERMINATE_HELD_MUTEX

7/2/2019 • 2 minutes to read • Edit Online

The THREAD_TERMINATE_HELD_MUTEX bug check has a value of 0x4000008A. This indicates that a driver acquired a mutex on a thread that exited before the mutex could be released. This can be caused by a driver returning to user mode without releasing a mutex or by a driver acquiring a mutex and then causing an exception that results in the thread it is running on, being terminated.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

THREAD_TERMINATE_HELD_MUTEX Parameters

PARAMETER	DESCRIPTION
1	The address of the KTHREAD that owns the KMUTEX.
2	The address of the KMUTEX that is owned.
3	Reserved
4	Reserved

Cause

To investigate, look at the callstack. If there is a driver on the stack that is directly followed by system exception handling routines and then thread termination routines, this driver is at fault and needs to be fixed so that it does not cause an unhandled exception while holding a kernel mutex. If the stack just shows normal thread termination code and no driver is implicated, run **!pool** or use **ln (List Nearest Symbols)** on the address of the mutex (parameter 2) and see if you can discover who owns the it. This bug will almost certainly be in the code of the owner of that mutex.

Bug Check 0xC0000218: STATUS_CANNOT_LOAD_REGISTRY_FILE

7/2/2019 • 2 minutes to read • Edit Online

The STATUS_CANNOT_LOAD_REGISTRY_FILE bug check has a value of 0xC0000218. This indicates that a registry file could not be loaded.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

STATUS_CANNOT_LOAD_REGISTRY_FILE Parameters

PARAMETER	DESCRIPTION
1	Address of the name of the registry hive that could not be loaded.
2	Zero (Reserved)
3	Zero (Reserved)
4	Zero (Reserved)

This bug check displays a descriptive text message. The name of the damaged file is displayed as part of the message.

Cause

This error occurs if a necessary registry hive file cannot be loaded. Usually this means the file is corrupt or is missing.

In rare instances, this error can be caused by a driver that has corrupted the registry image in memory, or by a memory error in this region.

Resolution

Try using the startup recovery mechanism (for example Startup Repair, Recovery Console, or Emergency Recovery Disk) provided by the operating system. If the problem is a missing or corrupt registry file, that usually fixes the problem.

Bug Check 0xC000021A: WINLOGON_FATAL_ERROR

10/15/2019 • 3 minutes to read • Edit Online

The WINLOGON_FATAL_ERROR bug check has a value of 0xC000021A. This means that that the Winlogon process terminated unexpectedly.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

WINLOGON_FATAL_ERROR Parameters

PARAMETER	DESCRIPTION
1	A string that identifies the problem
2	The error code
3	Reserved
4	Reserved

Cause

This error occurs when a user-mode subsystem, such as WinLogon or the Client Server Run-Time Subsystem (CSRSS), has been fatally compromised and security can no longer be guaranteed. In response, the operating system switches to kernel mode. Microsoft Windows cannot run without WinLogon or CSRSS. Therefore, this is one of the few cases where the failure of a user-mode service can shut down the system.

Mismatched system files can also cause this error. This mismatch can occur if you have restored your hard disk from a backup. Some backup programs might skip restoring system files that they determine are in use.

Resolution

Running the kernel debugger may not be useful in this situation because the actual error occurred in a user-mode process.

Resolving an error in a user-mode device driver, system service, or third-party application

Because bug check 0xC000021A occurs in a user-mode process, the most common culprits are third-party applications. If the error occurred after the installation of a new or updated device driver, system service, or third-party application, the new software should be removed or disabled to isolate the cause. Contact the manufacturer of the software about a possible update.

Resolving a mismatched system file problem

If you have recently restored your hard disk from a backup, check if there is an updated version of the

backup/restore program available from the manufacturer.

- Look at the most recently installed applications. To do this navigate to "Uninstall or change a program" in control panel and sort the installed applications by install date.
- Check the System Log in Event Viewer for additional error messages that might help pinpoint the device or driver that is causing the error. For more information, see Open Event Viewer. Look for critical errors in the system log that occurred in the same time window as the blue screen.
- Check the System Log in Event Viewer for additional error messages that might help pinpoint the device or driver that is causing the error. Look for critical errors in the system log that occurred in the same time window as the blue screen.

Remarks

- Use the System File Checker tool to repair missing or corrupted system files. The System File Checker is a utility in Windows that allows users to scan for corruptions in Windows system files and restore corrupted files. Use the following command to run the System File Checker tool (SFC.exe).

  ```
  SFC /scannow
  ```

  For more information, see Use the System File Checker tool to repair missing or corrupted system files.
- Run a virus detection program. Viruses can infect all types of hard disks that are formatted for Windows, and the resulting disk corruption can generate system bug check codes. Make sure the virus detection program checks the Master Boot Record for infections.
- Verify that the system has the latest updates installed. To detect which version is installed on your system, select **Start**, select **Run**, type **winver**, and then press ENTER. The **About Windows** dialog box displays the Windows version number (and the version number of the service pack, if one is installed).

Using Safe Mode

Consider using Safe Mode to isolate elements for troubleshooting and, if necessary, to use Windows. Using Safe Mode loads only the minimum required drivers and system services during the Windows startup.

To enter Safe Mode, use **Update and Security** in Settings. Select **Recovery** > **Advanced startup** to boot to maintenance mode. At the resulting menu, choose **Troubleshoot** > **Advanced Options** > **Startup Settings** > **Restart**. After Windows restarts and displays the **Startup Settings** screen, select option 4, 5, or 6 to boot to Safe Mode.

Safe Mode may also be available by pressing a function key on boot, for example F8. Refer to information from the computer's manufacturer for specific startup options.

Bug Check 0xC0000221: STATUS_IMAGE_CHECKSUM_MISMATCH

7/2/2019 • 3 minutes to read • Edit Online

The STATUS_IMAGE_CHECKSUM_MISMATCH bug check has a value of 0xC0000221. This indicates that a driver or a system DLL has been corrupted.

> **IMPORTANT**
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

STATUS_IMAGE_CHECKSUM_MISMATCH Parameters

Cause

This bug check results from a serious error in a driver or other system file. The file header checksum does not match the expected checksum.

This can also be caused by faulty hardware in the I/O path to the file (a disk error, faulty RAM, or a corrupted page file).

Resolution

To remedy this error, run the Emergency Recovery Disk (ERD) and allow the system to repair or replace the missing or damaged driver file on the system partition.

You can also run an in-place upgrade over the existing copy of Windows. This preserves all registry settings and configuration information, but replaces all system files. If any Service Packs and/or hotfixes had previously been applied, you need to reinstall them afterward in the appropriate order (latest Service Pack, then any post-Service Pack hotfixes in the order in which they were originally installed, if applicable).

If a specific file was identified in the bug check message as being corrupted, you can try replacing that individual file manually. If the system partition is formatted with FAT, you can start from an MS-DOS startup disk and copy the file from the original source onto the hard disk. If you have a dual-boot machine, you can boot to your other operating system and replace the file.

If you want to replace the file on a single-boot system with an NTFS partition, you need to restart the system, press F8 at the operating system **Loader** menu, and choose **Safe Mode with Command Prompt**. From there, copy a fresh version of the file from the original source onto the hard disk. If the file is used as part of the system startup process in Safe Mode, you need to start the computer using the Recovery Console in order to access the file. If these methods fail, try reinstalling Windows and then restoring the system from a backup.

Note If the original file from the product CD has a filename extension ending in an _ (underscore), the file needs to be uncompressed before it can be used. The Recovery Console's **Copy** command automatically detects compressed files and expands them as they are copied to the target location. If you are using Safe Mode to access a drive, use the **Expand** command to uncompress and copy the file to the target folder. You can use the **Expand** command in the command line environment of Safe Mode.

Resolving a disk error problem: Disk errors can be a source of file corruption. Run **Chkdsk /f /r** to detect and

resolve any file system structural corruption. You must restart the system before the disk scan begins on a system partition.

Resolving a RAM problem: If the error occurred immediately after RAM was added to the system, the paging file might be corrupted or the new RAM itself might be either faulty or incompatible.

To determine if newly added RAM is causing a bug check

1. Return the system to the original RAM configuration.
2. Use the Recovery Console to access the partition containing the paging file and delete the file pagefile.sys.
3. While still in the Recovery Console, run **Chkdsk /r** on the partition that contained the paging file.
4. Restart the system.
5. Set the paging file to an optimal level for the amount of RAM added.
6. Shutdown the system and add your RAM.

   The new RAM must meet the system manufacturer's specifications for speed, parity, and type (that is, fast page-mode (FPM) versus extended data out (EDO) versus synchronous dynamic random access memory (SDRAM)). Try to match the new RAM to the existing installed RAM as closely as possible. RAM can come in many different capacities, and more importantly, in different formats (single inline memory modules -- SIMM -- or dual inline memory modules -- DIMM). The electrical contacts can be either gold or tin and it is not wise to mix these contact types.

If you experience the same error message after reinstalling the new RAM, run hardware diagnostics supplied by the system manufacturer, especially the memory scanner. For details on these procedures, see the owner's manual for your computer.

When you can log on to the system again, check the System Log in Event Viewer for additional error messages that might help pinpoint the device or driver that is causing the error.

Disabling memory caching of the BIOS might also resolve this error.

Bug Check 0xDEADDEAD: MANUALLY_INITIATED_CRASH1

7/2/2019 • 2 minutes to read • Edit Online

The MANUALLY_INITIATED_CRASH1 bug check has a value of 0xDEADDEAD. This indicates that the user deliberately initiated a crash dump from either the kernel debugger or the keyboard.

> **IMPORTANT**
>
> This topic is for programmers. If you are a customer who has received a blue screen error code while using your computer, see Troubleshoot blue screen errors.

MANUALLY_INITIATED_CRASH1 Parameters

None

Remarks

For details on manually-initiated crash dumps, see Forcing a System Crash.

Debugger Reference

12/5/2018 • 2 minutes to read • Edit Online

This reference section includes:

Command-Line Options

Environment Variables

Debugger Commands

Debugger-Related APIs

Debugger Error and Warning Messages

WinDbg Graphical Interface Features

Command-Line Options

12/5/2018 • 2 minutes to read • Edit Online

In this section

- **CDB Command-Line Options**
- **KD Command-Line Options**
- **WinDbg Command-Line Options**
- **DbgSrv Command-Line Options**
- **KdSrv Command-Line Options**
- **DbEngPrx Command-Line Options**
- **KDbgCtrl Command-Line Options**
- **DbgRpc Command-Line Options**
- **SymStore Command-Line Options**

CDB Command-Line Options

1/3/2019 • 15 minutes to read • Edit Online

First-time users of CDB or NTSD should begin with the Debugging Using CDB and NTSD section.

The CDB command line uses the following syntax:

```
cdb  [ -server ServerTransport | -remote ClientTransport ]
[ -premote SmartClientTransport ] [-log{a|au|o|ou} LogFile]
[-2] [-d] [-ddefer] [-g] [-G] [-hd] [-lines] [-myob] [-bonc]
[-n] [-o] [-s] [-v] [-w] [-cf "filename"] [-cfr "filename"] [-c "command"]
[-robp] [-r BreakErrorLevel]  [-t PrintErrorLevel]
[ -x{e|d|n|i} Exception ] [-x] [-clines lines]
[-i ImagePath]  [-y SymbolPath] [-srcpath SourcePath]
[-aExtension] [-failinc] [-noio] [-noinh] [-noshell] [-nosqm]
[-sdce] [-ses] [-sicv] [-sins] [-snc] [-snul] [-zp PageFile]
[-sup] [-sflags 0xNumber] [-ee {masm|c++}]
[-e Event] [-pb] [-pd] [-pe] [-pr] [-pt Seconds] [-pv]
[ -- | -p PID | -pn Name | -psn ServiceName | -z DumpFile | executable ]
[-cimp] [-isd] [-kqm] [-pvr] [-version] [-vf] [-vf:<opts>] [-netsyms:{yes|no}]

cdb -iae

cdb -iaec KeyString

cdb -iu KeyString

cdb -QR Server

cdb -wake pid

cdb -?
```

The NTSD command-line syntax is identical to that of CDB:

```
ntsd  [ -server ServerTransport | -remote ClientTransport ]
[ -premote SmartClientTransport ] [-log{a|au|o|ou} LogFile]
[-2] [-d] [-ddefer] [-g] [-G] [-hd] [-lines] [-myob] [-bonc]
[-n] [-o] [-s] [-v] [-w] [-cf "filename"] [-cfr "filename"] [-c "command"]
[-robp] [-r BreakErrorLevel]  [-t PrintErrorLevel]
[ -x{e|d|n|i} Exception ] [-x] [-clines lines]
[-i ImagePath]  [-y SymbolPath] [-srcpath SourcePath]
[-aExtension] [-failinc] [-noio] [-noinh] [-noshell] [-nosqm]
[-sdce] [-ses] [-sicv] [-sins] [-snc] [-snul] [-zp PageFile]
[-sup] [-sflags 0xNumber] [-ee {masm|c++}]
[-e Event] [-pb] [-pd] [-pe] [-pr] [-pt Seconds] [-pv]
[ -- | -p PID | -pn Name | -psn ServiceName | -z DumpFile | executable ]
[-cimp] [-isd] [-kqm] [-pvr] [-version] [-vf] [-vf:<opts>] [-netsyms:{yes|no}]

ntsd -iae

ntsd -iaec KeyString

ntsd -iu KeyString

ntsd -QR Server

ntsd -wake PID

ntsd -?
```

The only difference between NTSD and CDB is that NTSD spawns a new console window while CDB inherits the window from which it was invoked. Since the **start** command can also be used to spawn a new console window, the following two constructions will give the same results:

```
start cdb [parameters]
ntsd [parameters]
```

Descriptions of the CDB and NTSD command-line options follow. Only the **-remote**, **-server**, **-g** and **-G** options are case-sensitive. The initial hyphen can be replaced with a forward-slash (/). Options that do not take any additional parameters can be concatenated -- so **cdb -o -d -G -g winmine** can be written as **cdb -odGg winmine**.

If the **-remote** or **-server** option is used, it must appear before any other options on the command line. If an *executable* is specified, it must appear last on the command line; any text after the *executable* name is passed to the executable program as its own command-line parameters.

Parameters

-server *ServerTransport*
Creates a debugging server that can be accessed by other debuggers. For an explanation of the possible *ServerTransport* values, see **Activating a Debugging Server**. When this parameter is used, it must be the first parameters on the command line.

-remote *ClientTransport*
Creates a debugging client, and connects to a debugging server that is already running. For an explanation of the possible *ClientTransport* values, see **Activating a Debugging Client**. When this parameter is used, it must be the first parameters on the command line.

-premote *SmartClientTransport*
Creates a smart client, and connects to a process server that is already running. For an explanation of the possible *SmartClientTransport* values, see **Activating a Smart Client**.

-2

If the target application is a *console application*, this option causes it to live in a new console window. (The default is for a target console application to share the window with CDB or NTSD.)

--
Debugs the Client Server Run-Time Subsystem (CSRSS). For details, see Debugging CSRSS.

-a *Extension*
Sets the default extension DLL. The default is *userexts*. There must be no space after the "a", and the .dll extension must not be included. For details, and other methods of setting this default, see Loading Debugger Extension DLLs.

-bonc
If this option is specified, the debugger will break into the target as soon as the session begins. This is especially useful when connecting to a debugging server that might not be currently broken into the target.

-c " *command* **"**
Specifies the initial debugger command to run at start-up. This command must be surrounded with quotation marks. Multiple commands can be separated with semicolons. (If you have a long command list, it may be easier to put them in a script and then use the **-c** option with the **$<, $><, $><, $$>< (Run Script File)** command.)

If you are starting a debugging client, this command must be intended for the debugging server. Client-specific commands such as **.lsrcpath** are not allowed.

-cf " *filename* **"**
Specifies the path and name of a script file. This script file is executed as soon as the debugger is started. If *filename* contains spaces it must be enclosed in quotation marks. If the path is omitted, the current directory is assumed. If the **-cf** option is not used, the file ntsd.ini in the current directory is used as the script file. If the file does not exist, no error occurs. For details, see Using Script Files.

-cfr " *filename* **"**
Specifies the path and name of a script file. This script file is executed as soon as the debugger is started, and any time the target is restarted. If *filename* contains spaces it must be enclosed in quotation marks. If the path is omitted, the current directory is assumed. If the file does not exist, no error occurs. For details, see Using Script Files.

-cimp
Directs CDB/NTSD to start with a DbgSrv implicit command line instead of an explicit process to run. This option is the client side of dbgsrv -pc.

-clines *lines*
Sets the approximate number of commands in the command history which can be accessed during remote debugging. For details, and for other ways to change this number, see Using Debugger Commands.

-d
Passes control of this debugger to the kernel debugger. If you are debugging CSRSS, this control redirection always is active, even if **-d** is not specified. (This option cannot be used during remote debugging -- use **-ddefer** instead.) See Controlling the User-Mode Debugger from the Kernel Debugger for details. This option cannot be used in conjunction with either the **-ddefer** option or the **-noio** option.

Note If you use WinDbg as the kernel debugger, many of the familiar features of WinDbg are not available in this scenario. For example, you cannot use the Locals window, the Disassembly window, or the Call Stack window, and you cannot step through source code. This is because WinDbg is only acting as a viewer for the debugger (NTSD or CDB) running on the target computer.

-ddefer
Passes control of this debugger to the kernel debugger, unless a debugging client is connected. (This is a

variation of **-d** that can be used from a debugging server.) See Controlling the User-Mode Debugger from the Kernel Debugger for details. This option cannot be used in conjunction with either the **-d** option or the **-noio** option.

-e *Event*
Signals the debugger that the specified event has occurred. This option is only used when starting the debugger programmatically.

-ee {**masm**|**c++**}
Sets the default expression evaluator. If **masm** is specified, MASM expression syntax will be used. If **c++** is specified, C++ expression syntax will be used. If the **-ee** option is omitted, MASM expression syntax is used as the default. See Evaluating Expressions for details.

-failinc
Causes the debugger to ignore any questionable symbols. When debugging a user-mode or kernel-mode minidump file, this option will also prevent the debugger from loading any modules whose images can't be mapped. For details and for other methods of controlling this, see SYMOPT_EXACT_SYMBOLS.

-g
Ignores the initial breakpoint in target application. This option will cause the target application to continue running after it is started or CDB attaches to it, unless another breakpoint has been set. See Initial Breakpoint for details.

-G
Ignores the final breakpoint at process termination. By default, CDB stops during the image run-down process. This option will cause CDB to exit immediately when the child terminates. This has the same effect as entering the command **sxd epr**. For more information, see Controlling Exceptions and Events.

-hd
Specifies that the debug heap should not be used. See Debugging a User-Mode Process Using CDB for details.

-i *ImagePath*
Specifies the location of the executables that generated the fault. If the path contains spaces, it should be enclosed in quotation marks.

-iae
Installs CDB as the postmortem debugger. For details, see Enabling Postmortem Debugging.

If this action succeeds, no message is displayed; if it fails, an error message is displayed.

The **-iae** parameter must not be used with any other parameters. This command will not actually start CDB.

-iaec *KeyString*
Installs CDB as the postmortem debugger. The contents of *KeyString* will be appended to the end of the **AeDebug** registry key. If *KeyString* contains spaces, it must be enclosed in quotation marks. For details, see Enabling Postmortem Debugging.

If this action succeeds, no message is displayed; if it fails, an error message is displayed.

The **-iaec** parameter must not be used with any other parameters. This command will not actually start CDB.

-isd
Turns on the CREATE_IGNORE_SYSTEM_DEFAULT flag for any process creations.

-iu *KeyString*
Registers debugger remoting as an URL type so that users can auto-launch a debugger remote client with an URL. *KeyString* has the format `remdbgeng://RemotingOption`. *RemotingOption* is a string that defines the

transport protocol as defined in the topic **Activating a Debugging Client**. If this action succeeds, no message is displayed; if it fails, an error message is displayed.

The **-iu** parameter must not be used with any other parameters. This command will not actually start CDB.

-kqm
Starts CDB/NTSD in quiet mode.

-lines
Enables source line debugging. If this option is omitted, the **.lines (Toggle Source Line Support)** command will have to be used before source debugging will be allowed. For other methods of controlling this, see SYMOPT_LOAD_LINES.

-log{**a**|**au**|**o**|**ou**} *LogFile*
Begins logging information to a log file. If the specified file already exists, it will be overwritten if **-logo** is used, or output will be appended to the file if -loga is used. The **-logau** and **-logou** options operate similar to **-loga** and **-logo** respectively, except that the log file is a Unicode file. For more details, see Keeping a Log File in CDB.

-myob
If there is a version mismatch with dbghelp.dll, the debugger will continue to run. (Without the **-myob** switch, this is considered a fatal error.)

-n
Noisy symbol load: Enables verbose output from the symbol handler. For details and for other methods of controlling this, see SYMOPT_DEBUG.

-netsyms {yes|no}
Allow or disallow loading symbols from a network path.

-noinh
Prevents processes created by the debugger from inheriting handles from the debugger. For other methods of controlling this, see Debugging a User-Mode Process Using CDB.

-noio
Prevents the debugging server from being used for input or output. Input will only be accepted from the debugging client (plus any initial command or command script specified by the **-c** command-line option).

All output will be directed to the debugging client. If NTSD is used for the server, no console window will be created at all. For more details, see **Activating a Debugging Server**. This option cannot be used in conjunction with either the **-d** option or the **-ddefer** option.

-noshell
Prohibits all **.shell** commands. This prohibition will last as long as the debugger is running, even if a new debugging session is begun. For details, and for other ways to disable **.shell** commands, see Using Shell Commands.

-nosqm
Disables telemetry data collection and upload.

-o
Debugs all processes launched by the target application (child processes). By default, processes created by the one you are debugging will run as they normally do. For other methods of controlling this, see Debugging a User-Mode Process Using CDB.

-p *PID*
Specifies the decimal process ID to be debugged. This is used to debug a process that is already running. For details, see Debugging a User-Mode Process Using CDB.

-pb
Prevents the debugger from requesting an initial break-in when attaching to a target process. This can be useful if the application is already suspended, or if you wish to avoid creating a break-in thread in the target.

-pd
Causes the target application not to be terminated at the end of the debugging session. See Ending a Debugging Session in CDB for details.

-pe
Indicates that the target application is already being debugged. See Re-attaching to the Target Application for details.

-pn *Name*
Specifies the name of the process to be debugged. (This name must be unique.) This is used to debug a process that is already running.

-pr
Causes the debugger to start the target process running when it attaches to it. This can be useful if the application is already suspended and you wish it to resume execution.

-psn *ServiceName*
Specifies the name of a service contained in the process to be debugged. This is used to debug a process that is already running.

-pt *Seconds*
Specifies the break time-out, in seconds. The default is 30. See Controlling the Target for details.

-pv
Specifies that the debugger should attach to the target process noninvasively. For details, see Noninvasive Debugging (User Mode).

-pvr
Works like **-pv** except that the target process is not suspended.

-QR *Server*
Lists all debugging servers running on the specified network server. The double backslash (\) preceding *Server* is optional. See **Searching for Debugging Servers** for details.

The **-QR** parameter cannot be used with any other parameters. This command will not actually start CDB.

-r *BreakErrorLevel*
Specifies the error level that will cause the target to break into the debugger. This is a decimal number equal to 0, 1, 2, or 3. Possible values are as follows:

VALUE	CONSTANT	MEANING
0	NONE	Do not break on any errors.
1	ERROR	Break on ERROR level debugging events.
2	MINORERROR	Break on MINORERROR and ERROR level debugging events.

VALUE	CONSTANT	MEANING
3	WARNING	Break on WARNING, MINORERROR, and ERROR level debugging events.

This error level only has meaning in checked builds of Microsoft Windows. The default value is 1.

-robp
This allows CDB to set a breakpoint on a read-only memory page. (The default is for such an operation to fail.)

-s
Disables lazy symbol loading. This will slow down process startup. For details and for other methods of controlling this, see SYMOPT_DEFERRED_LOADS.

-sdce
Causes the debugger to display **File access error** dialog boxes during symbol load. For details and for other methods of controlling this, see SYMOPT_FAIL_CRITICAL_ERRORS.

-ses
Causes the debugger to perform a strict evaluation of all symbol files and ignore any questionable symbols. For details and for other methods of controlling this, see SYMOPT_EXACT_SYMBOLS.

-sflags 0x *Number*
Sets all the symbol handler options at once. *Number* should be a hexadecimal number prefixed with **0x** -- a decimal without the **0x** is permitted, but the symbol options are binary flags and therefore hexadecimal is recommended. This option should be used with care, since it will override all the symbol handler defaults. For details, see Setting Symbol Options.

-sicv
Causes the symbol handler to ignore the CV record. For details and for other methods of controlling this, see SYMOPT_IGNORE_CVREC.

-sins
Causes the debugger to ignore the symbol path and executable image path environment variables. For details, see SYMOPT_IGNORE_NT_SYMPATH.

-snc
Causes the debugger to turn off C++ translation. For details and for other methods of controlling this, see SYMOPT_NO_CPP.

-snul
Disables automatic symbol loading for unqualified names. For details and for other methods of controlling this, see SYMOPT_NO_UNQUALIFIED_LOADS.

-srcpath *SourcePath*
Specifies the source file search path. Separate multiple paths with a semicolon (;). If the path contains spaces, it should be enclosed in quotation marks. For details, and for other ways to change this path, see Source Path.

-sup
Causes the symbol handler to search the public symbol table during every symbol search. For details and for other methods of controlling this, see SYMOPT_AUTO_PUBLICS.

-t *PrintErrorLevel*
Specifies the error level that will cause the debugger to display an error message. This is a decimal number equal to 0, 1, 2, or 3. Possible values are as follows:

VALUE	CONSTANT	MEANING
0	NONE	Do not display any errors.
1	ERROR	Display ERROR level debugging events.
2	MINORERROR	Display MINORERROR and ERROR level debugging events.
3	WARNING	Display WARNING, MINORERROR, and ERROR level debugging events.

This error level only has meaning in checked builds of Microsoft Windows. The default value is 1.

-v
Enables verbose output from the debugger.

-version
Prints the debugger version string.

-vf
Enables default ApplicationVerifier settings.

-vf: *<opts>*
Enables given ApplicationVerifier settings.

-w
Specifies to debug 16-bit applications in a separate VDM.

-wake *PID*
Causes sleep mode to end for the user-mode debugger whose process ID is specified by *PID*. This command must be issued on the target machine during sleep mode. See Controlling the User-Mode Debugger from the Kernel Debugger for details.

The **-wake** parameter should not be used with any other parameters. This command will not actually start CDB.

-x{**e**|**d**|**n**|**i**} *Exception*
Controls the debugger's behavior when the specified event occurs. The *Exception* can be either an exception number or an event code. You can specify this option multiple times to control different events. See Controlling Exceptions and Events for details and for other methods of controlling these settings.

-x
Disables first-chance break on access violation exceptions. The second occurrence of an access violation will break into the debugger. This is the same as **-xd av**.

-y *SymbolPath*
Specifies the symbol search path. Separate multiple paths with a semicolon (;). If the path contains spaces, it should be enclosed in quotation marks. For details, and for other ways to change this path, see Symbol Path.

-z *DumpFile*
Specifies the name of a crash dump file to debug. If the path and file name contain spaces, this must be surrounded by quotation marks. It is possible to open several dump files at once by including multiple **-z**

options, each followed by a different *DumpFile* value. For details, see Analyzing a User-Mode Dump File.

-zp *PageFile*
Specifies the name of a modified page file. This is useful if you are debugging a dump file and want to use the **.pagein (Page In Memory)** command. You cannot use **-zp** with a standard Windows page file -- only specially-modified page files can be used.

executable
Specifies the command line of an executable process. This is used to launch a new process and debug it. This has to be the final item on the command line. All text after the executable name is passed to the executable as its argument string.

-?
Displays command-line help text.

When you are starting the debugger from **Start | Run** or from a Command Prompt window, specify arguments for the target application after the application's file name. For instance:

```
cdb myexe arg1arg2
```

KD Command-Line Options

1/3/2019 • 9 minutes to read • Edit Online

First-time users of KD should begin with the Debugging Using KD and NTKD section.

The KD command line uses the following syntax.

```
kd [ -server ServerTransport | -remote ClientTransport ]
   [-b | -x] [-d] [-bonc] [-m] [-myob] [-lines] [-n] [-r] [-s]
   [-v] [-clines lines] [-failinc] [-noio] [-noshell]
   [-secure] [-sdce] [-ses] [-sicv] [-sins] [-snc] [-snul]
   [-sup] [-sflags 0xNumber] [-log{a|au|o|ou} LogFile]
   [-aExtension] [-zp PageFile]
   [-i ImagePath] [-y SymbolPath]  [-srcpath SourcePath]
   [-k ConnectType | -kl | -kqm | -kx ExdiOptions] [-ee {masm|c++}]
   [-z DumpFile] [-cf "filename"] [-cfr "filename"] [-c "command"]
   [-t PrintErrorLevel] [-version]

kd -iu KeyString

kd -QR Server

kd -wake PID

kd -?
```

Descriptions of the KD command-line options follow. Only the **-remote** and **-server** options are case-sensitive. The initial hyphen can be replaced with a forward-slash (/). Options which do not take any additional parameters can be concatenated -- so **kd -r -n -v** can be written as **kd -rnv**.

If the **-remote** or **-server** option is used, it must appear before any other options on the command line.

Parameters

-server *ServerTransport*
Creates a debugging server that can be accessed by other debuggers. For an explanation of the possible *ServerTransport*, see **Activating a Debugging Server**. When this parameter is used, it must be the first parameters on the command line.

-remote *ClientTransport*
Creates a debugging client, and connects to a debugging server that is already running. For an explanation of the possible *ClientTransport* values, see **Activating a Debugging Client**. When this parameter is used, it must be the first parameters on the command line.

-a *Extension*
Sets the default extension DLL. The default is kdextx86.dll or kdexts.dll. There must be no space after the "a", and the .dll file name extension must not be included. For details, and other methods of setting this default, see Loading Debugger Extension DLLs.

-b
This option no longer supported.

-bonc
If this option is specified, the debugger will break into the target as soon as the session begins. This is especially useful when connecting to a debugging server that might not be currently broken into the target.

-c "*command*"
Specifies the initial debugger command to run at start-up. This command must be surrounded with quotation marks. Multiple commands can be separated with semicolons. (If you have a long command list, it may be easier to put them in a script and then use the **-c** option with the **$<, $><, $><, $$>< (Run Script File)** command.)

If you are starting a debugging client, this command must be intended for the debugging server. Client-specific commands, such as **.lsrcpath**, are not allowed.

-cf "*filename*"
Specifies the path and name of a script file. This script file is executed as soon as the debugger is started. If *filename* contains spaces it must be enclosed in quotation marks. If the path is omitted, the current directory is assumed. If the -cf option is not used, the file ntsd.ini in the current directory is used as the script file. If the file does not exist, no error occurs. For details, see Using Script Files.

-cfr "*filename*"
Specifies the path and name of a script file. This script file is executed as soon as the debugger is started, and any time the target is restarted. If *filename* contains spaces it must be enclosed in quotation marks. If the path is omitted, the current directory is assumed. If the file does not exist, no error occurs. For details, see Using Script Files.

-clines *lines*
Sets the approximate number of commands in the command history which can be accessed during remote debugging. For details, and for other ways to change this number, see Using Debugger Commands.

-d
After a reboot, the debugger will break into the target computer as soon as a kernel module is loaded. (This break is earlier than the break from the **-b** option.) See Crashing and Rebooting the Target Computer for details and for other methods of changing this status.

-ee {**masm**|**c++**}
Sets the default expression evaluator. If **masm** is specified, MASM expression syntax will be used. If **c++** is specified, C++ expression syntax will be used. If the **-ee** option is omitted, MASM expression syntax is used as the default. See Evaluating Expressions for details.

-failinc
Causes the debugger to ignore any questionable symbols. When debugging a user-mode or kernel-mode minidump file, this option will also prevent the debugger from loading any modules whose images can't be mapped. For details and for other methods of controlling this, see SYMOPT_EXACT_SYMBOLS.

-i *ImagePath*
Specifies the location of the executables that generated the fault. If the path contains spaces, it should be enclosed in quotation marks.

-iu *KeyString*
Registers debugger remoting as an URL type so that users can auto-launch a debugger remote client with an URL. *KeyString* has the format `remdbgeng://RemotingOption`. *RemotingOption* is a string that defines the transport protocol as defined in the topic **Activating a Debugging Client**. If this action succeeds, no message is displayed; if it fails, an error message is displayed.

The **-iu** parameter must not be used with any other parameters. This command will not actually start KD.

-k *ConnectType*
Tells the debugger how to connect to the target. For details, see Debugging Using KD and NTKD.

-kl
Starts a kernel debugging session on the same machine as the debugger.

-kqm

Starts KD in quiet mode.

-kx *ExdiOptions*
Starts a kernel debugging session using an EXDI driver. EXDI drivers are not described in this documentation. If you have an EXDI interface to your hardware probe or hardware simulator, please contact Microsoft for debugging information.

-lines
Enables source line debugging. If this option is omitted, the **.lines (Toggle Source Line Support)** command will have to be used before source debugging will be allowed. For other methods of controlling this, see SYMOPT_LOAD_LINES.

-log{**a**|**au**|**o**|**ou**} *LogFile*
Begins logging information to a log file. If *LogFile* already exists, it will be overwritten if **-logo** is used, or output will be appended to the file if **-loga** is used. The **-logau** and **-logou** options operate similar to **-loga** and **-logo** respectively, except that the log file is a Unicode file. For more details, see Keeping a Log File in KD.

-m
Indicates that the serial port is connected to a modem. Instructs the debugger to watch for the carrier-detect signal.

-myob
If there is a version mismatch with dbghelp.dll, the debugger will continue to run. (Without the **-myob** switch, this is considered a fatal error.)

A secondary effect of this option is that the warning that normally appears when breaking into the target computer is suppressed.

-n
Noisy symbol load: Enables verbose output from symbol handler. For details and for other methods of controlling this, see SYMOPT_DEBUG.

-noio
Prevents the debugging server from being used for input or output. Input will only be accepted from the debugging client (plus any initial command or command script specified by the **-c** command-line option).

All output will be directed to the debugging client. For more details, see **Activating a Debugging Server**.

-noshell
Prohibits all **.shell** commands. This prohibition will last as long as the debugger is running, even if a new debugging session is begun. For details, and for other ways to disable shell commands, see Using Shell Commands.

-QR *Server*
Lists all debugging servers running on the specified network server. The double backslash (****) preceding *Server* is optional. See **Searching for Debugging Servers** for details.

The **-QR** parameter must not be used with any other parameters. This command will not actually start KD.

-r
Displays registers.

-s
Disables lazy symbol loading. This will slow down process startup. For details and for other methods of controlling this, see SYMOPT_DEFERRED_LOADS.

-sdce
Causes the debugger to display **File access error** dialog boxes during symbol load. For details and for other

methods of controlling this, see SYMOPT_FAIL_CRITICAL_ERRORS.

-secure
Activates Secure Mode.

-ses
Causes the debugger to perform a strict evaluation of all symbol files and ignore any questionable symbols. For details and for other methods of controlling this, see SYMOPT_EXACT_SYMBOLS.

-sflags 0x*Number*
Sets all the symbol handler options at once. *Number* should be a hexadecimal number prefixed with **0x** -- a decimal without the **0x** is permitted, but the symbol options are binary flags and therefore hexadecimal is recommended. This option should be used with care, since it will override all the symbol handler defaults. For details, see Setting Symbol Options.

-sicv
Causes the symbol handler to ignore the CV record. For details and for other methods of controlling this, see SYMOPT_IGNORE_CVREC.

-sins
Causes the debugger to ignore the symbol path and executable image path environment variables. For details, see SYMOPT_IGNORE_NT_SYMPATH.

-snc
Causes the debugger to turn off C++ translation. For details and for other methods of controlling this, see SYMOPT_NO_CPP.

-snul
Disables automatic symbol loading for unqualified names. For details and for other methods of controlling this, see SYMOPT_NO_UNQUALIFIED_LOADS.

-srcpath *SourcePath*
Specifies the source file search path. Separate multiple paths with a semicolon (**;**). If the path contains spaces, it should be enclosed in quotation marks. For details, and for other ways to change this path, see Source Path.

-sup
Causes the symbol handler to search the public symbol table during every symbol search. For details and for other methods of controlling this, see SYMOPT_AUTO_PUBLICS.

-t *PrintErrorLevel*
Specifies the error level that will cause the debugger to display an error message. This is a decimal number equal to 0, 1, 2, or 3. The values are described as follows:

VALUE	CONSTANT	MEANING
0	NONE	Do not display any errors.
1	ERROR	Display ERROR level debugging events.
2	MINORERROR	Display MINORERROR and ERROR level debugging events.

VALUE	CONSTANT	MEANING
3	WARNING	Display WARNING, MINORERROR, and ERROR level debugging events.

This error level only has meaning in checked builds of Microsoft Windows. The default value is 1.

-v
Generates verbose messages for loads, deferred loads, and unloads.

-version
Prints the debugger version string.

-wake *PID*
Causes sleep mode to end for the user-mode debugger whose process ID is specified by *PID*. This command must be issued on the target machine during sleep mode. See Controlling the User-Mode Debugger from the Kernel Debugger for details.

The **-wake** parameter must not be used with any other parameters. This command will not actually start KD.

-x
Causes the debugger to break in when an exception first occurs, rather than letting the application or module that caused the exception deal with it. (Same as **-b**, except with an initial **eb nt!NtGlobalFlag 9;g** command.)

-y *SymbolPath*
Specifies the symbol search path. Separate multiple paths with a semicolon (**;**). If the path contains spaces, it should be enclosed in quotation marks. For details, and for other ways to change this path, see Symbol Path.

-z *DumpFile*
Specifies the name of a crash dump file to debug. If the path and file name contain spaces, this must be surrounded by quotation marks. It is possible to open several dump files at once by including multiple **-z** options, each followed by a different *DumpFile* value. For details, see Analyzing a Kernel-Mode Dump File with KD.

-zp *PageFile*
Specifies the name of a modified page file. This is useful if you are debugging a dump file and want to use the **.pagein (Page In Memory)** command. You cannot use **-zp** with a standard Windows page file—only specially-modified page files can be used.

-?
Displays command-line help text.

KD will automatically detect the platform on which the target is running. You do not need to specify the target on the KD command line. The older syntax (using the name *I386KD* or *IA64KD*) is obsolete.

WinDbg Command-Line Options

12/5/2018 • 11 minutes to read • Edit Online

First-time users of WinDbg should begin with the Debugging Using WinDbg section.

The WinDbg command line uses the following syntax:

```
windbg [ -server ServerTransport | -remote ClientTransport ] [-lsrcpath ]
   [ -premote SmartClientTransport ] [-?] [-ee {masm|c++}]
   [-clines lines] [-b] [-d] [-aExtension]
   [-failinc] [-g] [-G] [-hd] [-j] [-n] [-noshell] [-o] [-openPrivateDumpByHandle Handle]
   [-Q | -QY] [-QS | -QSY] [-robp] [-secure] [-ses] [-sdce]
   [-sicv] [-sins] [-snc] [-snul] [-sup] [-sflags 0xNumber]
   [-T Title] [-v] [-log{o|a} LogFile] [-noinh]
   [-i ImagePath] [-y SymbolPath] [-srcpath SourcePath]
   [-k [ConnectType] | -kl | -kx ExdiOptions] [-c "command"]
   [-pb] [-pd] [-pe] [-pr] [-pt Seconds] [-pv]
   [-W Workspace] [-WF Filename] [-WX] [-zp PageFile]
   [ -p PID | -pn Name | -psn ServiceName | -z DumpFile | executable ]

windbg -I[S]

windbg -IU KeyString

windbg -IA[S]
```

Descriptions of the WinDbg command-line options follow. All command-line options are case-sensitive except for **-j**. The initial hyphen can be replaced with a forward-slash (/).

If the **-remote** or **-server** option is used, it must appear before any other options on the command line. If an *executable* is specified, it must appear last on the command line; any text after the *executable* name is passed to the executable program as its own command-line parameters.

Parameters

-server *ServerTransport*
Creates a debugging server that can be accessed by other debuggers. For an explanation of the possible *ServerTransport* values, see **Activating a Debugging Server**. When this parameter is used, it must be the first parameters on the command line.

-remote *ClientTransport*
Creates a debugging client, and connects to a debugging server that is already running. For an explanation of the possible *ClientTransport* values, see **Activating a Debugging Client**. When this parameter is used, it must be the first parameters on the command line.

-premote *SmartClientTransport*
Creates a smart client, and connects to a process server that is already running. For an explanation of the possible *SmartClientTransport* values, see **Activating a Smart Client**.

-a *Extension*
Sets the default extension DLL. The default is kdextx86.dll or kdexts.dll. There must be no space after the "a", and the .dll file name extension must not be included. For details, and other methods of setting this default, see Loading Debugger Extension DLLs.

-b

This option is no longer supported.

-c " *command* "
Specifies the initial debugger command to run at start-up. This command must be enclosed in quotation marks. Multiple commands can be separated with semicolons. (If you have a long command list, it may be easier to put them in a script and then use the **-c** option with the **$<, $><, $><, $$>< (Run Script File)** command.)

If you are starting a debugging client, this command must be intended for the debugging server. Client-specific commands, such as **.lsrcpath**, are not allowed.

-clines *lines*
Sets the approximate number of commands in the command history which can be accessed during remote debugging. For details, and for other ways to change this number, see Using Debugger Commands.

-d
(Kernel mode only) After a reboot, the debugger will break into the target computer as soon as a kernel module is loaded. (This break is earlier than the break from the **-b** option.) See Crashing and Rebooting the Target Computer for details and for other methods of changing this status.

-ee {**masm**|**c++**}
Sets the default expression evaluator. If **masm** is specified, MASM expression syntax will be used. If **c++** is specified, C++ expression syntax will be used. If the **-ee** option is omitted, MASM expression syntax is used as the default. See Evaluating Expressions for details.

-failinc
Causes the debugger to ignore any questionable symbols. When debugging a user-mode or kernel-mode minidump file, this option will also prevent the debugger from loading any modules whose images can't be mapped. For details and for other methods of controlling this, see SYMOPT_EXACT_SYMBOLS.

-g
(User mode only) Ignores the initial breakpoint in target application. This option will cause the target application to continue running after it is started or WinDbg attaches to it, unless another breakpoint has been set. See Initial Breakpoint for details.

-G
(User mode only) Ignores the final breakpoint at process termination. Typically, the debugging session ends during the image run-down process. This option will cause the debugging session to end immediately when the child terminates. This has the same effect as entering the command **sxd epr**. For more information, see Controlling Exceptions and Events.

-hd
(User mode only) Specifies that the debug heap should not be used.

-I[**S**]
Installs WinDbg as the postmortem debugger. For details, see Enabling Postmortem Debugging.

After this action is attempted, a success or failure message is displayed. If **S** is included, this procedure is done silently if it is successful; only failure messages are displayed.

The **-I** parameter must not be used with any other parameters. This command will not actually start WinDbg, although a WinDbg window may appear for a moment.

-IA[**S**]
Associates WinDbg with the file extensions .dmp, .mdmp, and .wew in the registry. After this action is attempted, a success or failure message is displayed. If **S** is included, this procedure is done silently if it is successful; only failure messages are displayed. After this association is made, double-clicking a file with one of these extensions will start WinDbg.

The **-IA** parameter must not be used with any other parameters. This command will not actually start WinDbg, although a WinDbg window may appear for a moment.

-IU *KeyString*
Registers debugger remoting as an URL type so that users can auto-launch a debugger remote client with an URL. *KeyString* has the format `remdbgeng://RemotingOption`. *RemotingOption* is a string that defines the transport protocol as defined in the topic **Activating a Debugging Client**. If this action succeeds, no message is displayed; if it fails, an error message is displayed.

The **-IU** parameter must not be used with any other parameters. Although a WinDbg window may appear for a moment, this command will not actually start WinDbg.

-i *ImagePath*
Specifies the location of the executables that generated the fault. If the path contains spaces, it should be enclosed in quotation marks.

-j
Allow journaling.

-k [*ConnectType*]
(Kernel mode only) Starts a kernel debugging session. For details, see Live Kernel-Mode Debugging Using WinDbg. If **-k** is used without any *ConnectType* options following it, it must be the final entry on the command line.

-kl
(Kernel mode only) Starts a kernel debugging session on the same machine as the debugger.

-kx *ExdiOptions*
(Kernel mode only) Starts a kernel debugging session using an EXDI driver. EXDI drivers are not described in this documentation. If you have an EXDI interface to your hardware probe or hardware simulator, please contact Microsoft for debugging information.

-log{**o**|**a**} *LogFile*
Begins logging information to a log file. If the specified log file already exists, it will be overwritten if **-logo** is used. If **loga** is used, the output will be appended to the file. For more details, see Keeping a Log File in WinDbg.

-lsrcpath
Sets the local source path for a remote client. This option must follow **-remote** on the command line.

-n
Noisy symbol load: Enables verbose output from symbol handler. For details and for other methods of controlling this, see SYMOPT_DEBUG.

-noinh
(User mode only) Prevents processes created by the debugger from inheriting handles from the debugger. For other methods of controlling this, see Debugging a User-Mode Process Using WinDbg.

-noprio
Prevents any priority change. This parameter will prevent WinDbg from taking priority for CPU time while active.

-noshell
Prohibits all **.shell** commands. This prohibition will last as long as the debugger is running, even if a new debugging session is begun. For details, and for other ways to disable shell commands, see Using Shell Commands.

-o

(User mode only) Debugs all processes launched by the target application (child processes). By default, processes created by the one you are debugging will run as they normally do.

-openPrivateDumpByHandle *Handle*
Specifies the handle of a crash dump file to debug.

-p *PID*
Specifies the decimal process ID to be debugged. This is used to debug a process that is already running.

-pb
(User mode only) Prevents the debugger from requesting an initial break-in when attaching to a target process. This can be useful if the application is already suspended, or if you wish to avoid creating a break-in thread in the target.

-pd
(User mode only) Causes the target application not to be terminated at the end of the debugging session. See Ending a Debugging Session in WinDbg for details.

-pe
(User mode only) Indicates that the target application is already being debugged. See Re-attaching to the Target Application for details.

-pn *Name*
Specifies the name of the process to be debugged. (This name must be unique.) This is used to debug a process that is already running.

-pr
(User mode only) Causes the debugger to start the target process running when it attaches to it. This can be useful if the application is already suspended and you wish it to resume execution.

-psn *ServiceName*
Specifies the name of a service contained in the process to be debugged. This is used to debug a process that is already running.

-pt *Seconds*
Specifies the break timeout, in seconds. The default is 30. See Controlling the Target for details.

-pv
(User mode only) Specifies that the debugger should attach to the target process noninvasively. For details, see Noninvasive Debugging (User Mode).

-Q
Suppresses the "Save Workspace?" dialog box. Workspaces are not automatically saved. See Using Workspaces for details.

-QS
Suppresses the "Reload Source?" dialog box. Source files are not automatically reloaded.

-QSY
Suppresses the "Reload Source?" dialog box and automatically reloads source files.

-QY
Suppresses the "Save Workspace?" dialog box and automatically saves workspaces. See Using Workspaces for details.

-robp
This allows CDB to set a breakpoint on a read-only memory page. (The default is for such an operation to fail.)

-sdce
Causes the debugger to display **File access error** messages during symbol load. For details and for other methods of controlling this, see SYMOPT_FAIL_CRITICAL_ERRORS.

-secure
Activates Secure Mode.

-ses
Causes the debugger to perform a strict evaluation of all symbol files and ignore any questionable symbols. For details and for other methods of controlling this, see SYMOPT_EXACT_SYMBOLS.

-sflags 0x *Number*
Sets all the symbol handler options at once. *Number* should be a hexadecimal number prefixed with **0x** -- a decimal without the **0x** is permitted, but the symbol options are binary flags and therefore hexadecimal is recommended. This option should be used with care, since it will override all the symbol handler defaults. For details, see Setting Symbol Options.

-sicv
Causes the symbol handler to ignore the CV record. For details and for other methods of controlling this, see SYMOPT_IGNORE_CVREC.

-sins
Causes the debugger to ignore the symbol path and executable image path environment variables. For details, see SYMOPT_IGNORE_NT_SYMPATH.

-snc
Causes the debugger to turn off C++ translation. For details and for other methods of controlling this, see SYMOPT_NO_CPP.

-snul
Disables automatic symbol loading for unqualified names. For details and for other methods of controlling this, see SYMOPT_NO_UNQUALIFIED_LOADS.

-srcpath *SourcePath*
Specifies the source file search path. Separate multiple paths with a semicolon (**;**). If the path contains spaces, it should be enclosed in quotation marks. For details, and for other ways to change this path, see Source Path.

-sup
Causes the symbol handler to search the public symbol table during every symbol search. For details and for other methods of controlling this, see SYMOPT_AUTO_PUBLICS.

-T *Title*
Sets WinDbg window title.

-v
Enables verbose output from debugger.

-W *Workspace*
Loads the given named workspace. If the workspace name contains spaces, enclose it in quotation marks. If no workspace of this name exists, you will be given the option of creating a new workspace with this name or abandoning the load attempt. For details, see Using Workspaces.

-WF *Filename*
Loads the workspace from the given file. *Filename* should include the file and the extension (usually .wew). If the workspace name contains spaces, enclose it in quotation marks. If no workspace file with this name exists, you will be given the option of creating a new workspace file with this name or abandoning the load attempt. For details, see Using Workspaces.

-WX
Disables automatic workspace loading. For details, see Using Workspaces.

-y *SymbolPath*
Specifies the symbol search path. Separate multiple paths with a semicolon (**;**). If the path contains spaces, it should be enclosed in quotation marks. For details, and for other ways to change this path, see Symbol Path.

-z *DumpFile*
Specifies the name of a crash dump file to debug. If the path and file name contain spaces, this must be surrounded by quotation marks. It is possible to open several dump files at once by including multiple **-z** options, each followed by a different *DumpFile* value. For details, see Analyzing a User-Mode Dump File or Analyzing a Kernel-Mode Dump File with WinDbg.

-zp *PageFile*
Specifies the name of a modified page file. This is useful if you are debugging a dump file and want to use the **.pagein (Page In Memory)** command. You cannot use **-zp** with a standard Windows page file -- only specially-modified page files can be used.

executable
Specifies the command line of an executable process. This is used to launch a new process and debug it. This has to be the final item on the command line. All text after the executable name is passed to the executable as its argument string. For details, see Debugging a User-Mode Process Using WinDbg.

-?
Pops up this HTML Help window.

When you are running the debugger from the command line, specify arguments for the target application after application's file name. For instance:

```
windbg myexe arg1 arg2
```

DbgSrv Command-Line Options

7/1/2019 • 2 minutes to read • Edit Online

The DbgSrv command line uses the following syntax.

```
dbgsrv -t ServerTransport [-sifeo image.ext] -c[s] AppCmdLine [-x | -pc]

dbgsrv -?
```

All options are case-sensitive.

Parameters

-t *ServerTransport*
Specifies the transport protocol. For a list of the possible protocols and the syntax for *ServerTransport* in each case, see **Activating a Process Server**.

-sifeo *Executable*
Suspends the Image File Execution Option (IFEO) value for the given image. *Executable* should include the file name of the executable image, including the file name extensions. The -sifeo option allows DbgSrv to be set as the IFEO debugger for an image created by the -c option, without causing recursive invocation due to the IFEO setting. This option can be used only if -c is used.

-c
Causes DbgSrv to create a new process. You can use this to create a process that you intend to debug. This is similar to spawning a new process from the debugger, except that this process will *not* be debugged when it is created. To debug this process, determine its PID and use the **-p** option when starting the smart client to debug this process.

s
Causes the newly-created process to be immediately suspended. If you are using this option, it is recommended that you use CDB as your smart client, and that you start the smart client with the -pb command-line option, in conjunction with -p PID. If you include the -pb option on the command line, the process will resume when the debugger attaches to it; otherwise you can resume the process with the **~*m** command.

AppCmdLine
Specifies the full command line of the process to be created. *AppCmdLine* can be either a Unicode or ASCII string, and can include any printable character. All text that appears after the **-c**[**s**] parameter will be taken to form the string *AppCmdLine*.

-x
Causes the remainder of the command line to be ignored. This option is useful if you are launching DbgSrv from an application that may append unwanted text to its command line.

-pc
Causes the remainder of the command line to be used as the "implicit command line" as a pending process creation. This command line will be used if a debugger is connected to this process server with the "-cimp". For instance, running `dbgsrv -t <ServerTransport> -pc notepad.exe`, and then running `ntsd -premote <Transport> -cimp` will cause ntsd to connect to dbgsrv and launch notepad.exe

-?
Displays a message box with help text for the DbgSrv command line.

For information about using DbgSrv, see Process Servers (User Mode).

KdSrv Command-Line Options

12/5/2018 • 2 minutes to read • Edit Online

The KdSrv command line uses the following syntax.

```
kdsrv -t ServerTransport
```

Parameters

-t *ServerTransport*
Specifies the transport protocol. For a list of the possible protocols and the syntax for *ServerTransport* in each case, see **Activating a KD Connection Server**.

If you type **kdsrv** with no parameters, a message box with help text appears.

For information about using KdSrv, see KD Connection Servers (Kernel Mode).

DbEngPrx Command-Line Options

12/5/2018 • 2 minutes to read • Edit Online

The DbEngPrx command line uses the following syntax.

```
dbengprx [-p] -c ClientTransport -s ServerTransport

dbengprx -?
```

Parameters

-p
Causes DbEngPrx to continue existing even after all connections to it are dropped.

-c *ClientTransport*
Specifies the protocol settings to be used in connecting to the server. The protocol should match that used when the server was created. For details, see **Activating a Repeater**.

-s *ServerTransport*
Specifies the protocol settings that will be used when the client connects to the repeater. For details, see **Activating a Repeater**.

-?
Displays a message box with help text for the DbEngPrx command line.

For information about using DbEngPrx, see Repeaters.

KDbgCtrl Command-Line Options

12/5/2018 • 2 minutes to read • Edit Online

The KDbgCtrl command line uses the following syntax:

```
kdbgctrl [-e|-d|-c] [-ea|-da|-ca] [-eu|-du|-cu] [-eb|-db|-cb] [-sdb Size | -cdb]

kdbgctrl -cx

kdbgctrl -td ProcessID File

kdbgctrl -?
```

Parameters

-e
Enables Full Kernel Debugging.

-d
Disables Full Kernel Debugging.

-c
Checks whether Full Kernel Debugging is enabled. Displays true if Full Kernel Deubugging is enabled, and displays false if Full Kernel Debugging is disabled.

-ea
Enables Automatic Kernel Debugging.

-da
Disables Automatic Kernel Debugging.

-ca
Checks whether Automatic Kernel Debugging is enabled. Displays true if Automatic Kernel Deubugging is enabled, and displays false if Automatic Kernel Debugging is disabled.

-eu
Enables User-Mode Error Handling.

-du
Disables User-Mode Error Handling.

-cu
Checks whether User-Mode Error Handling is enabled. Displays true if User-Mode Error Handling is enabled, and displays false if User-Mode Error Handling is disabled.

-eb
Enables blocking of kernel debugging.

-db
Disables blocking of kernel debugging

-cb
Checks whether kernel debugging is blocked. Displays true if kernel debugging is blocked, and displays false if kernel debugging is not blocked.

-sdb *Size*
Sets the size of the DbgPrint buffer. If *Size* is prefixed with **0x** it will be interpreted as a hexadecimal number. If it is prefixed with **0** (zero), it will be interpreted as octal. Otherwise, it will be interpreted as decimal.

-cdb
Displays the current size, in bytes, of the DbgPrint buffer.

-cx
Determines the current Full Kernel Debugging setting and returns an appropriate value. This option cannot be combined with other options, and it does not display any output. It is designed for use in a batch file where the return value of the KDbgCtrl program can be tested. Possible return values are as follows:

VALUE	MEANING
0x10001	Full Kernel Debugging is enabled.
0x10002	Full Kernel Debugging is disabled.
Any other value	An error occurred. KDbgCtrl was unable to determine the current status of Full Kernel Debugging.

-td *ProcessID File*
Obtains a kernel triage dump file. Enter the process ID and a name for the dump file.

-?
Displays command-line help for KDbgCtrl.

Additional Information

For a description of all the KDbgCtrl settings, see Using KDbgCtrl.

DbgRpc Command-Line Options

12/5/2018 • 2 minutes to read • Edit Online

The DbgRpc command line must always contain exactly one of the -l, -e, -t, -c, or -a switches. The options following these switches depend on the switch used. The -s, -p, and -r options can be used with any other options.

```
 dbgrpc [-s Server -p ProtSeq] [-r Radix] -l -P ProcessID -L CellID1.CellID2

dbgrpc [-s Server -p ProtSeq] [-r Radix] -e [-E EndpointName]

dbgrpc [-s Server -p ProtSeq] [-r Radix] -t -P ProcessID [-T ThreadID]

dbgrpc [-s Server -p ProtSeq] [-r Radix] [-c|-a] [-C CallID] [-I IfStart] [-N ProcNum] [-P ProcessID]

dbgrpc -?
```

Parameters

-s *Server*
Allows DbgRpc to view information from a remote machine. The server name should not be preceded by slash marks. For more information about using DbgRpc remotely, see Using the DbgRpc Tool.

-p *ProtSeq*
Specifies the remote transport to be used. The possible values of *ProtSeq* are **ncacn_ip_tcp** (TCP protocol) and **ncacn_np** (named pipe protocol). TCP protocol is recommended. For more information about using DbgRpc remotely, see Using the DbgRpc Tool.

-r *Radix*
Specifies the radix to be used for the command parameters. The default is base 16. If the **-r** parameter is used, it should be placed first on the line, since it only affects parameters listed after itself. It does not affect the output of the DbgRpc tool.

-l
Displays RPC state information for the specified cell. For an example, see Get RPC Cell Information.

ProcessID
Specifies the process ID (PID) of a process. When the **-l** option is being used, this should be the process whose server contains the desired cell. When the **-t** option is being used, this should be the process containing the desired thread. When the **-c** or **-a** options are being used, this parameter is optional; it should be the server process that owns the calls you wish to display.

CellID1.CellID2
Specifies the number of the cell to be displayed.

-e
Searches the system's RPC state information for endpoint information. For an example, see Get RPC Endpoint Information.

EndpointName
Specifies the number of the endpoint to be displayed. If omitted, the endpoints for all processes on the system are displayed.

-t

Searches the system's RPC state information for thread information. For an example, see Get RPC Thread Information.

ThreadID
Specifies the thread ID of the thread to be displayed. If omitted, all threads in the specified process will be displayed.

-c
Searches the system's RPC state information for server-side call (SCALL) information. For an example, see Get RPC Call Information.

-a
Searches the system's RPC state information for client call (CCALL) information. For an example, see Get RPC Client Call Information. This option requires full RPC state information.

CallID
Specifies the call ID. This parameter is optional; include it only if you want to display calls matching a specific *CallID* value.

IfStart
Specifies the first DWORD of the interface's universally unique identifier (UUID) on which the call was made. This parameter is optional; include it only if you want to display calls matching a specific *IfStart* value.

ProcNum
Specifies the procedure number of this call. (The RPC Run-Time identifies individual routines from an interface by numbering them by position in the IDL file -- the first routine in the interface is 0, the second 1, and so on.) This parameter is optional; include it only if you want to display calls matching a specific *ProcNum* value.

Additional Information

For more information about debugging Microsoft Remote Procedure Call (RPC), see RPC Debugging.

SymStore Command-Line Options

3/14/2019 • 3 minutes to read • Edit Online

The following syntax forms are supported for SymStore transactions. The first parameter is **add**, **query** or **del**. Use **/?** to display the available options.

```
symstore add [/r] [/p [/l] [-:MSG Message] [-:REL] [-:NOREFS]] /f File /s Store /t Product [/v Version] [/o]
[/c Comment] [/d LogFile] [/compress]

symstore add [/r] [/p [/l] [-:REL] [-:NOREFS]] /g Share /f File /x IndexFile [/a] [/o] [/d LogFile]

symstore add /y IndexFile /g Share /s Store [/p [-:MSG Message] [-:REL] [-:NOREFS]] /t Product [/v Version]
[/o] [/c Comment] [/d LogFile] [/compress]

symstore query [/r] /f File /s Store [/o] [/d LogFile]

symstore del /i ID /s Store [/o] [/d LogFile]

symstore /?
```

Parameters

/f *File*
Specifies the network path of files or directories to add.

/g *Share*
Specifies the server and share where the symbol files were originally stored. When used with **/f**, *Share* should be identical to the beginning of the *File* specifier. When used with **/y**, *Share* should be the location of the original symbol files (not the index file). This allows you to later change this portion of the file path in case you move the symbol files to a different server and share.

/s *Store*
Specifies the root directory for the symbol store.

/m *Prefix*
Causes SymStore to prefer using symbols from paths beginning with *Prefix* when storing files or updating pointers. This option cannot be used with the **/x** option.

/h { **PUB | PRI** }
Causes SymStore to prefer using public symbols (if PUB is specified), or private symbols (if PRI is specified) when storing or updating symbols. This option has no effect on binary files.

/i *ID*
Specifies the transaction ID string.

/p
Causes SymStore to store a pointer to the file, rather than the file itself.

/l
Allows the file specified by *File* to be in a local directory rather than a network path. (This option can only be used when both **/f** and **/p** are used.)

-:MSG *Message*
Adds the specified *Message* to each file. (This option can only be used when **/p** is used.)

-:REL
Allows the paths in the file pointers to be relative. This option implies the **/l** option. (This option can only be used when **/p** is used.)

-:NOREFS
Omits the creation of reference pointer files for the files and pointers being stored. This option is only valid during the initial creation of a symbol store if the store being changed was created with this option.

/r
Causes SymStore to add files or directories recursively.

/t *Product*
Specifies the name of the product.

/v *Version*
Specifies the version of the product.

/c *Comment*
Specifies a comment for the transaction.

/d *LogFile*
Specifies a log file to be used for command output. If this is not included, transaction information and other output is sent to **stdout**.

/o
Causes SymStore to display verbose output.

/x *IndexFile*
Causes SymStore not to store the actual symbol files. Instead, SymStore records information in the *IndexFile* that will enable SymStore to access the symbol files at a later time.

/a
Causes SymStore to append new indexing information to an existing index file. (This option is only used with the **/x** option.)

/y *IndexFile*
Causes SymStore to read the data from a file created with **/x**.

/yi *IndexFile*
Appends a comment with the transaction ID to the end of an index file created with the /**x** option.

/z { **PUB** | **PRI** }
Causes SymStore to index only the type of symbols specified. If **PUB** is specified, then only the symbols that have had the full source information stripped will be indexed. If **PRI** is specified, then only the symbols that contain the full source information will be indexed. SymStore will always index binary symbols.

/compress
Causes SymStore to create a compressed version of each file copied to the symbol store instead of using an uncompressed copy of the file. This option is only valid when storing files and not pointers, and consequently cannot be used when the **/p** option is used.

/?
Displays help text for the SymStore command.

Additional Information

For more information about SymStore, see Using Symbol Servers and Symbol Stores.

Environment Variables

12/5/2018 • 2 minutes to read • Edit Online

This reference section includes:

General Environment Variables

Kernel-Mode Environment Variables

For information about using environment variables for debugging, see Getting Set Up for Debugging.

General Environment Variables

1/3/2019 • 2 minutes to read • Edit Online

The following table lists the environment variables that can be used in both user-mode and kernel-mode debugging.

VARIABLE	MEANING
_NT_DEBUGGER_EXTENSION_PATH = *Path*	Specifies the path that the debugger will first search for extension DLLs. *Path* can contain a drive letter followed by a colon (**:**). Separate multiple directories with semicolons (**;**). For details, see Loading Debugger Extension DLLs.
_NT_EXECUTABLE_IMAGE_PATH = *Path*	Specifies the path containing the binary executable files. *Path* can contain a drive letter followed by a colon (**:**). Separate multiple directories with semicolons (**;**).
_NT_SOURCE_PATH = *Path*	Specifies the path containing the source files for the target. *Path* can contain a drive letter followed by a colon (**:**). Separate multiple directories with semicolons (**;**). For details, and for other ways to change this path, see Source Path.
_NT_SYMBOL_PATH = *Path*	Specifies the root of a directory tree containing the symbol files. *Path* can contain a drive letter followed by a colon (**:**). Separate multiple directories with semicolons (**;**). For details, and for other ways to change this path, see Symbol Path.
_NT_ALT_SYMBOL_PATH = *Path*	Specifies an alternate symbol path searched before _NT_SYMBOL_PATH. This is useful for keeping private versions of symbol files. *Path* can contain a drive letter followed by a colon (**:**). Separate multiple directories with semicolons (**;**). For details, see Symbol Path.
_NT_SYMBOL_PROXY = *Proxy***:***Port*	Specifies the proxy server to be used by SymSrv. For details, see Firewalls and Proxy Servers.
_NT_DEBUG_HISTORY_SIZE = *Number*	Specifies the number of commands in the command history that can be accessed during remote debugging. Because commands vary in length, the number of lines available may not exactly match *Number*. For details, and for other ways to change this number, see Using Debugger Commands.
_NT_DEBUG_LOG_FILE_OPEN = *Filename*	(CDB and KD only) Specifies the log file to which the debugger should send output.

VARIABLE	MEANING
_NT_DEBUG_LOG_FILE_APPEND = *Filename*	(CDB and KD only) Specifies the log file to which the debugger should append output.
_NT_EXPR_EVAL = {**masm** | **c++**}	Specifies the default expression evaluator. If **masm** is specified, MASM expression syntax will be used. If **c++** is specified, C++ expression syntax will be used. MASM expression syntax is the default. See Evaluating Expressions for details.
_NO_DEBUG_HEAP	Specifies that the debug heap should not be used for user-mode debugging.
DBGENG_NO_DEBUG_PRIVILEGE	Prevents processes spawned by the debugger from inheriting SeDebugPrivilege.
DBGENG_NO_BUGCHECK_ANALYSIS	Prevents automated bugcheck analysis.
DBGHELP_HOMEDIR	Specifies the path for the root of the default downstream store used by SymSrv and SrcSrv. *Path* can contain a drive letter followed by a colon (**:**). Separate multiple directories with semicolons (**;**).
SRCSRV_INI_FILE	Specifies the path and name of the configuration file used by SrcSrv. By default, the path is the srcsrv subdirectory of the Debugging Tools for Windows installation directory, and the file name is Srcsrv.ini. See Source Indexing for details.

Kernel-Mode Environment Variables

12/5/2018 • 2 minutes to read • Edit Online

The following table lists the environment variables that are used only in kernel-mode debugging.

VARIABLE	MEANING
_NT_DEBUG_PORT = *ComPort*	Specifies the COM port to be used in a kernel connection. For details, see Getting Set Up for Debugging.
_NT_DEBUG_BAUD_RATE = *BaudRate*	Specifies the baud rate to be used over the COM port connection.
_NT_DEBUG_BUS = 1394	Specifies that kernel debugging will be done over a 1394 cable connection.
_NT_DEBUG_1394_CHANNEL = *1394Channel*	Specifies the channel to be used for the 1394 kernel connection.
_NT_DEBUG_1394_SYMLINK = *Protocol*	Specifies the connection protocol to be used for the 1394 kernel connection.
KDQUIET =*Anything*	If KDQUIET is defined, the debugger will run in *quiet mode*. Quiet mode involves three distinct effects: 1. The debugger does not display messages each time an extension DLL is loaded or unloaded. 2. The **r (Registers)** command no longer requires an equal sign in its syntax. 3. The debugger will not display a warning message when breaking into the target computer. Quiet mode can also be controlled by using the **sq (Set Quiet Mode)** command.
_NT_DEBUG_CACHE_SIZE = *Size*	Specifies the maximum kernel debugging cache size, in bytes. This cache holds data received by the host computer from the serial connection. The default is 1,024,000.

VARIABLE	MEANING
_NT_DEBUG_OPTIONS = *Option*	Specifies one of the following two values: NOEXTWARNING tells the debugger not to output a warning when it cannot find an extension command. NOVERSIONCHECK tells the debugger not to check the version of debugger extensions. These options can be modified or displayed by using the **so (Set Kernel Options)** command.
_NT_KD_FILES = *MapFile*	Specifies a driver replacement map file. For details and for other methods of controlling driver replacement, see Mapping Driver Files.

Debugger Commands

12/5/2018 • 2 minutes to read • Edit Online

This section includes the following topics:

Syntax Rules

Command Tokens

Commands

Meta-Commands

Control Keys

Syntax Rules

12/5/2018 • 2 minutes to read • Edit Online

This section describes the syntax rules that you must follow to use debugger commands.

When you are debugging, you should obey the following general syntax rules:

- You can use any combination of uppercase and lowercase letters in commands and arguments, except when specifically noted in the topics in this section.
- You can separate multiple command parameters by one or more spaces or by a comma (**,**).
- You can typically omit the space between a command and its first parameter . You can frequently omit other spaces if this omission does not cause any ambiguity.

The command reference topics in this section use the following items:

- Characters in **bold** font style indicate items that you must literally type.
- Characters in *italic* font style indicate parameters that are explained in the "Parameters" section of the reference topic.
- Parameters in brackets (**[***xxx***]**) are optional. Brackets with a vertical bar (**[***xxx***|***yyy***]**) indicate that you can use one, or none, of the enclosed parameters.
- Braces with vertical bars (**{***xxx***|***yyy***}**) indicate that you must use exactly one of the enclosed parameters .

The following topics describe the syntax that the following parameter types use:

Numerical Expression Syntax

String Wildcard Syntax

Register Syntax

Pseudo-Register Syntax

Source Line Syntax

Address and Address Range Syntax

Thread Syntax

Process Syntax

System Syntax

Multiprocessor Syntax

Syntax also plays an important role in using symbols. For further details, see Symbol Syntax and Symbol Matching.

Numerical Expression Syntax

12/5/2018 • 2 minutes to read • Edit Online

The debugger accepts two different kinds of numeric expressions: *C++ expressions* and *MASM expressions*. Each of these expressions follows its own syntax rules for input and output.

For more information about when each syntax type is used, see Evaluating Expressions.

This section includes the following topics:

MASM Numbers and Operators

C++ Numbers and Operators

MASM Expressions vs. C++ Expressions

Expression Examples

Sign Extension

MASM Numbers and Operators

6/17/2019 • 7 minutes to read • Edit Online

Before version 4.0 of the Debugging Tools for Windows package, NTSD, CDB, KD, and WinDbg used only Microsoft Macro Assembler (MASM) expression syntax.

Numbers in MASM Expressions

You can put numbers in MASM expressions in base 16, 10, 8, or 2.

Use the **n (Set Number Base)** command to set the default radix to 16, 10, or 8. All unprefixed numbers are then interpreted in this base. You can override the default radix by specifying the **0x** prefix (hexadecimal), the **0n** prefix (decimal), the **0t** prefix (octal), or the **0y** prefix (binary).

You can also specify hexadecimal numbers by adding an **h** after the number. You can use uppercase or lowercase letters within numbers. For example, "0x4AB3", "0X4aB3", "4AB3h", "4ab3h", and "4aB3H" have the same meaning.

If you do not add a number after the prefix in an expression, the number is read as 0. Therefore, you can write 0 as 0, the prefix followed by 0, and only the prefix. For example, in hexadecimal, "0", "0x0", and "0x" have the same meaning.

You can enter hexadecimal 64-bit values in the **xxxxxxxx`xxxxxxxx** format. You can also omit the grave accent (`). If you include the grave accent, automatic sign extension is disabled.

Symbols in MASM Expressions

In MASM expressions, the numeric value of any symbol is its memory address. Depending on what the symbol refers to, this address is the address of a global variable, local variable, function, segment, module, or any other recognized label.

To specify which module the address is associated with, include the module name and an exclamation point (!) before the name of the symbol. If the symbol could be interpreted as a hexadecimal number, include the module name and an exclamation point, or just an exclamation point, before the symbol name. For more information about symbol recognition, see Symbol Syntax and Symbol Matching.

Use two colons (::) or two underscores (__) to indicate the members of a class.

Use a grave accent (`) or an apostrophe (') in a symbol name only if you add a module name and exclamation point before the symbol.

Numeric Operators in MASM Expressions

You can modify any component of an expression by using a unary operator. You can combine any two components by using a binary operator. Unary operators take precedence over binary operators. When you use multiple binary operators, the operators follow the fixed precedence rules that are described in the following tables.

You can always use parentheses to override precedence rules.

If part of an MASM expression is enclosed in parentheses and two at signs (@@) appear before the expression, the expression is interpreted according to C++ expression rules. You cannot add a space between the two at signs and the opening parenthesis. You can also specify the expression evaluator by using **@@c++(...)** or **@@masm(...)**.

When you perform arithmetic computations, the MASM expression evaluator treats all numbers and symbols as

ULONG64 types.

Unary address operators assume DS as the default segment for addresses. Expressions are evaluated in order of operator precedence. If adjacent operators have equal precedence, the expression is evaluated from left to right.

You can use the following unary operators.

OPERATOR	MEANING
+	Unary plus
-	Unary minus
not	Returns 1 if the argument is zero. Returns zero for any nonzero argument.
hi	High 16 bits
low	Low 16 bits
by	Low-order byte from the specified address.
$pby	Same as **by** except that it takes a physical address. Only physical memory that uses the default caching behavior can be read.
wo	Low-order word from the specified address.
$pwo	Same as **wo** except that it takes a physical address. Only physical memory that uses the default caching behavior can be read.
dwo	Double-word from the specified address.
$pdwo	Same as **dwo** except that it takes a physical address. Only physical memory that uses the default caching behavior can be read.
qwo	Quad-word from the specified address.
$pqwo	Same as **qwo** except that it takes a physical address. Only physical memory that uses the default caching behavior can be read.

OPERATOR	MEANING
poi	Pointer-sized data from the specified address. The pointer size is 32 bits or 64 bits. In kernel debugging, this size is based on the processor of the *target* computer. In user-mode debugging on an Itanium-based computer, this size is 32 bits or 64 bits, depending on the target application. Therefore, **poi** is the best operator to use if you want pointer-sized data.
$ppoi	Same as **poi** except that it takes a physical address. Only physical memory that uses the default caching behavior can be read.

You can use the following binary operators. The operators in each cell take precedence over those in lower cells. Operators in the same cell are of the same precedence and are parsed from left to right.

OPERATOR	MEANING
*	Multiplication
/	Integer division
mod (or %)	Modulus (remainder)
+	Addition
-	Subtraction
< <	Left shift
> >	Logical right shift
> > >	Arithmetic right shift
= (or ==)	Equal to
<	Less than
>	Greater than
< =	Less than or equal to
> =	Greater than or equal to
!=	Not equal to
and (or &)	Bitwise AND
xor (or ^)	Bitwise XOR (exclusive OR)
or (or |)	Bitwise OR

The <, >, =, ==, and != comparison operators evaluate to 1 if the expression is true or zero if the expression is false. A single equal sign (=) is the same as a double equal sign (==). You cannot use side effects or assignments

within a MASM expression.

An invalid operation (such as division by zero) results in an "Operand error" is returned to the Debugger Command window.

Non-Numeric Operators in MASM Expressions

You can also use the following additional operators in MASM expressions.

OPERATOR	MEANING
$fnsucc(*FnAddress*, *RetVal*, *Flag***)**	Interprets the *RetVal* value as a return value for the function that is located at the *FnAddress* address. If this return value qualifies as a success code, **$fnsucc** returns **TRUE**. Otherwise, **$fnsucc** returns **FALSE**. If the return type is BOOL, bool, HANDLE, HRESULT, or NTSTATUS, **$fnsucc** correctly understands whether the specified return value qualifies as a success code. If the return type is a pointer, all values other than **NULL** qualify as success codes. For any other type, success is defined by the value of *Flag*. If *Flag* is 0, a nonzero value of *RetVal* is success. If *Flag* is 1, a zero value of *RetVal* is success.
$iment (*Address***)**	Returns the address of the image entry point in the loaded module list. *Address* specifies the Portable Executable (PE) image base address. The entry is found by looking up the image entry point in the PE image header of the image that *Address* specifies. You can use this function for both modules that are already in the module list and to set unresolved breakpoints by using the **bu** command.
$scmp("*String1***", "***String2***")**	Evaluates to -1, 0, or 1, like the **strcmp** C function.
$sicmp("*String1***", "***String2***")**	Evaluates to -1, 0, or 1, like the **stricmp** Microsoft Win32 function .
$spat("*String***", "***Pattern***")**	Evaluates to **TRUE** or **FALSE** depending on whether *String* matches *Pattern*. The matching is case-insensitive. *Pattern* can contain a variety of wildcard characters and specifiers. For more information about the syntax, see String Wildcard Syntax.
$vvalid(*Address***,** *Length***)**	Determines whether the memory range that begins at *Address* and extends for *Length* bytes is valid. If the memory is valid, **$vvalid** evaluates to 1. If the memory is invalid, **$vvalid** evaluates to 0.

Registers and Pseudo-Registers in MASM Expressions

You can use registers and pseudo-registers within MASM expressions. You can add an at sign (@) before all registers and pseudo-registers. The at sign causes the debugger to access the value more quickly. This at sign is unnecessary for the most common x86-based registers. For other registers and pseudo-registers, we recommend that you add the at sign, but it is not actually required. If you omit the at sign for the less common

registers, the debugger tries to parse the text as a hexadecimal number, then as a symbol, and finally as a register.

You can also use a period (.) to indicate the current instruction pointer. You should not add an at sign before this period, and you cannot use a period as the first parameter of the **r command**. This period has the same meaning as the **$ip** pseudo-register.

For more information about registers and pseudo-registers, see Register Syntax and Pseudo-Register Syntax.

Source Line Numbers in MASM Expressions

You can use source file and line number expressions within MASM expressions. You must enclose these expressions by using grave accents (`). For more information about the syntax, see Source Line Syntax.

C++ Numbers and Operators

6/17/2019 • 6 minutes to read • Edit Online

The C++ expression parser supports all forms of C++ expression syntax. The syntax includes all data types (including pointers, floating-point numbers, and arrays) and all C++ unary and binary operators.

Numbers in C++ Expressions

Numbers in C++ expressions are interpreted as decimal numbers, unless you specify them in another manner. To specify a hexadecimal integer, add **0x** before the number. To specify an octal integer, add**0** (zero) before the number.

The default debugger radix does not affect how you enter C++ expressions. You cannot directly enter a binary number (except by nesting a MASM expression within the C++ expression).

You can enter a hexadecimal 64-bit value in the *xxxxxxxx`xxxxxxxx* format. (You can also omit the grave accent ( `).) Both formats produce the same value.

You can use the **L**, **U**, and **I64** suffixes with integer values. The actual size of the number that is created depends on the suffix and the number that you enter. For more information about this interpretation, see a C++ language reference.

The *output* of the C++ expression evaluator keeps the data type that the C++ expression rules specify. However, if you use this expression as an argument for a command, a cast is always made. For example, you do not have to cast integer values to pointers when they are used as addresses in command arguments. If the expression's value cannot be validly cast to an integer or a pointer, a syntax error occurs.

You can use the **0n** (decimal) prefix for some *output*, but you cannot use it for C++ expression input.

Characters and Strings in C++ Expressions

You can enter a character by surrounding it with single quotation marks ('). The standard C++ escape characters are permitted.

You can enter string literals by surrounding them with double quotation marks ("). You can use **\"** as an escape sequence within such a string. However, strings have no meaning to the expression evaluator.

Symbols in C++ Expressions

In a C++ expression, each symbol is interpreted according to its type. Depending on what the symbol refers to, it might be interpreted as an integer, a data structure, a function pointer, or any other data type. If you use a symbol that does not correspond to a C++ data type (such as an unmodified module name) within a C++ expression, a syntax error occurs.

If the symbol might be ambiguous, you can add a module name and an exclamation point (**!**) or only an exclamation point before the symbol. For more information about symbol recognition, see Symbol Syntax and Symbol Matching.

You can use a grave accent (`) or an apostrophe (**'**) in a symbol name only if you add a module name and exclamation point before the symbol name.

When you add the **<** and **>** delimiters after a template name, you can add spaces between these delimiters.

Operators in C++ Expressions

You can always use parentheses to override precedence rules.

If you enclose part of a C++ expression in parentheses and add two at signs (**@@**) before the expression, the

expression is interpreted according to MASM expression rules. You cannot add a space between the two at signs and the opening parenthesis. The final value of this expression is passed to the C++ expression evaluator as a ULONG64 value. You can also specify the expression evaluator by using **@@c++(...)** or **@@masm(...)**.

Data types are indicated as usual in the C++ language. The symbols that indicate arrays (**[ ]**), pointer members (**->**), UDT members (**.**), and members of classes (**::**) are all recognized. All arithmetic operators are supported, including assignment and side-effect operators. However, you cannot use the **new**, **delete**, and **throw** operators, and you cannot actually call a function.

Pointer arithmetic is supported and offsets are scaled correctly. Note that you cannot add an offset to a function pointer. (If you have to add an offset to a function pointer, cast the offset to a character pointer first.)

As in C++, if you use operators with invalid data types, a syntax error occurs. The debugger's C++ expression parser uses slightly more relaxed rules than most C++ compilers, but all major rules are enforced. For example, you cannot shift a non-integer value.

You can use the following operators. The operators in each cell take precedence over those in lower cells. Operators in the same cell are of the same precedence and are parsed from left to right. As with C++, expression evaluation ends when its value is known. This ending enables you to effectively use expressions such as **?? myPtr && *myPtr**.

OPERATOR	MEANING
Expression **//** *Comment*	Ignore all subsequent text
Class **::** *Member* *Class* **::~***Member* **::** *Name*	Member of class Member of class (destructor) Global
Structure **.** *Field* *Pointer* **->** *Field* *Name* **[***integer***]** *LValue* **++** *LValue* **--** **dynamic_cast <***type***>(***Value***)** **static_cast <***type***>(***Value***)** **reinterpret_cast <***type***>(***Value***)** **const_cast <***type***>(***Value***)**	Field in a structure Field in referenced structure Array subscript Increment (after evaluation) Decrement (after evaluation) Typecast (always performed) Typecast (always performed) Typecast (always performed) Typecast (always performed)

OPERATOR	MEANING
(*type***)** *Value*	Typecast (always performed)
sizeof *value*	Size of expression
sizeof(*type* **)**	Size of data type
++ *LValue*	Increment (before evaluation)
-- *LValue*	Decrement (before evaluation)
~ *Value*	Bit complement
! *Value*	Not (Boolean)
Value	Unary minus
+ *Value*	Unary plus
& *LValue*	Address of data type
Value	Dereference
Structure **.** *Pointer*	Pointer to member of structure
Pointer **->** ***** *Pointer*	Pointer to member of referenced structure
Value Value	Multiplication
Value **/** *Value*	Division
Value **%** *Value*	Modulus
Value **+** *Value*	Addition
Value **-** *Value*	Subtraction
Value **<<** *Value*	Bitwise shift left
Value **>>** *Value*	Bitwise shift right
Value **<** *Value*	Less than (comparison)
Value **<=** *Value*	Less than or equal (comparison)
Value **>** *Value*	Greater than (comparison)
Value **>=** *Value*	Greater than or equal (comparison)
Value **==** *Value*	Equal (comparison)
Value **!=** *Value*	Not equal (comparison)
Value **&** *Value*	Bitwise AND
Value **^** *Value*	Bitwise XOR (exclusive OR)

OPERATOR	MEANING
Value **|** *Value*	Bitwise OR
Value **&&** *Value*	Logical AND
Value **||** *Value*	Logical OR
LValue **=***Value* *LValue* **=** *Value* *LValue* **/=** *Value* *LValue* **%=***Value* *LValue* **+=***Value* *LValue* **-=** *Value* *LValue* **<<=** *Value* *LValue* **>>=** *Value* *LValue* **&=** *Value* *LValue* **|=** *Value* *LValue* **^=** *Value*	Assign Multiply and assign Divide and assign Modulo and assign Add and assign Subtract and assign Shift left and assign Shift right and assign AND and assign OR and assign XOR and assign
Value **?** *Value* **:** *Value*	Conditional evaluation
Value **,** *Value*	Evaluate all values, and then discard all except the rightmost value

Registers and Pseudo-Registers in C++ Expressions

You can use registers and pseudo-registers within C++ expressions. You must add an at sign (**@**) before the register or pseudo-register.

The expression evaluator automatically performs the proper cast. Actual registers and integer-value pseudo-registers are cast to ULONG64. All addresses are cast to PUCHAR, **$thread** is cast to ETHREAD*, **$proc** is cast to EPROCESS*, **$teb** is cast to TEB*, and **$peb** is cast to PEB*.

You cannot change a register or pseudo-register by an assignment or side-effect operator. You must use the **r (Registers)** command to change these values.

For more information about registers and pseudo-registers, see Register Syntax and Pseudo-Register Syntax.

Macros in C++ Expressions

You can use macros within C++ expressions. You must add a number sign (#) before the macros.

You can use the following macros. These macros have the same definitions as the Microsoft Windows macros with the same name. (The Windows macros are defined in Winnt.h.)

MACRO	RETURN VALUE
#CONTAINING_RECORD(*Address*, *Type*, *Field*)	Returns the base address of an instance of a structure, given the type of the structure and the address of a field within the structure.
#FIELD_OFFSET(*Type*, *Field*)	Returns the byte offset of a named field in a known structure type.
#RTL_CONTAINS_FIELD (*Struct*, *Size*, *Field*)	Indicates whether the given byte size includes the desired field.
#RTL_FIELD_SIZE(*Type*, *Field*)	Returns the size of a field in a structure of known type, without requiring the type of the field.
#RTL_NUMBER_OF(*Array*)	Returns the number of elements in a statically sized array.
#RTL_SIZEOF_THROUGH_FIELD(*Type*, *Field*)	Returns the size of a structure of known type, up through and including a specified field.

MASM Expressions vs. C++ Expressions

12/5/2018 • 2 minutes to read • Edit Online

The most significant differences between MASM expression evaluation and C++ expression evaluation are as follows:

- In an MASM expression, the numeric value of any symbol is its memory address. In a C++ expression, the numeric value of a variable is its actual value, not its address. Data structures do not have numeric values. Instead, they are treated as actual structures and you must use them accordingly. The value of a function name or any other entry point is the memory address and is treated as a function pointer. If you use a symbol that does not correspond to a C++ data type (such as an unmodified module name), a syntax error occurs.
- The MASM expression evaluator treats all numbers as ULONG64 values. The C++ expression evaluator casts numbers to ULONG64 and preserves type information of all data types.
- The MASM expression evaluator lets you to use any operator together with any number. The C++ expression evaluator generates an error if you use an operator together with an incorrect data type.
- In the MASM expression evaluator, all arithmetic is performed literally. In the C++ expression evaluator, pointer arithmetic is scaled properly and is not permitted when inappropriate.
- An MASM expression can use two underscores (__) or two colons (**::**) to indicate members of a class. The C++ expression evaluator uses only the two-colon syntax. Debugger *output* always uses two colons.
- In an MASM expression, you should add an at sign (**@**) before all except the most common registers. If you omit this at sign, the register name might be interpreted as a hexadecimal number or as a symbol. In a C++ expression, this prefix is required for all registers.
- MASM expressions might contain references to source lines. These references are indicated by grave accents (`). You cannot reference source line numbers in a C++ expression.

Expression Examples

12/5/2018 • 2 minutes to read • Edit Online

This topics contains examples of MASM and C++ expressions that are used in various commands.

All other sections of this Help documentation use MASM expression syntax in the examples (unless otherwise noted). C++ expression syntax is very useful for manipulating structures and variables, but it does not parse the parameters of debugger commands very well.

If you are using debugger commands for general purposes or using debugger extensions, you should set MASM expression syntax as the default syntax. If you must have a specific parameter to use C++ expression syntax, use the **@@( )** syntax.

Conditional Breakpoints

You can use comparison operators to create conditional breakpoints. The following code example uses MASM expression syntax. Because the current default radix is 16, the example uses the **0n** prefix so that the number 20 is understood as a decimal number.

```
0:000> bp MyFunction+0x43 "j ( poi(MyVar)>0n20 ) ''; 'gc' "
```

In the previous example, **MyVar** is an integer in the C source. Because the MASM parser treats all symbols as addresses, the example must have the **poi** operator to dereference **MyVar**.

Conditional Expressions

The following example prints the value of **ecx** if **eax** is greater than **ebx**, 7 if **eax** is less than **ebx**, and 3 if **eax** equals **ebx**. This example uses the MASM expression evaluator, so the equal sign (=) is a comparison operator, not an assignment operator.

```
0:000> ? ecx*(eax>ebx) + 7*(eax<ebx) + 3*(eax=ebx)
```

In C++ syntax, the **@** sign indicates a register, a double equal sign (==) is the comparison operator, and code must explicitly cast from BOOL to **int**. Therefore, in C++ syntax, the previous command becomes the following.

```
0:000> ?? @ecx*(int)(@eax>@ebx) + 7*(int)(@eax<@ebx) + 3*(int)(@eax==@ebx)
```

C++ Expression Examples

If **myInt** is a ULONG32 value and if you are using the MASM expression evaluator, the following two examples show the value of **MyInt**.

```
0:000> ?? myInt
0:000> dd myInt L1
```

However, the following example shows the *address* of **myInt**.

```
0:000> ? myInt
```

Mixed Expression Examples

You cannot use source-line expressions in a C++ expression. The following example uses the **@@( )** syntax to nest an MASM expression within a C++ expression. This example sets **MyPtr** equal to the address of line 43 of the Myfile.c file.

```
0:000> ?? MyPtr = @@( `myfile.c:43` )
```

The following examples set the default expression evaluator to MASM and then evaluate *Expression2* as a C++ expression, and evaluate *Expression1* and *Expression3* as MASM expressions.

```
0:000> .expr /s masm
0:000> bp Expression1 + @@( Expression2 ) + Expression3
```

If **myInt** is a ULONG64 value and if you know that this value is followed in memory by another ULONG64, you can set an access breakpoint at that location by using one of the following examples. (Note the use of pointer arithmetic.)

```
0:000> ba r8 @@( &myInt + 1 )
0:000> ba r8 myInt + 8
```

Structures

The C++ expression evaluator casts pseudo-registers to their appropriate types. For example, **$teb** is cast as a TEB*. The following example displays the process ID.

```
kd> ??  @$teb->ClientId.UniqueProcess
```

Sign Extension

12/5/2018 • 2 minutes to read • Edit Online

When a 32-bit signed integer is negative, its highest bit is equal to one. When this 32-bit signed integer is cast to a 64-bit number, the high bits can be set to zero (preserving the unsigned integer and hexadecimal value of the number) or the high bits can be set to one (preserving the signed value of the number). The latter situation is called *sign extension*.

The debugger follows different rules for sign extension in MASM expressions, in C++ expressions, and when displaying numbers.

Sign Extension in MASM Expressions

Under certain conditions, numbers are automatically *sign extended* by the MASM expression evaluator. Sign extension can affect only numbers from 0x80000000 through 0xFFFFFFFF. That is, sign extension affects only numbers that can be written in 32 bits with the high bit equal to 1.

The number 0x12345678 always remains 0x00000000`12345678 when the debugger treats it as a 64-bit number. On the other hand, when 0x890ABCDE is treated as a 64-bit value, it might remain 0x00000000`890ABCDE or the MASM expression evaluator might sign extend it to 0xFFFFFFFF`890ABCDE.

A number from 0x80000000 through 0xFFFFFFFF is sign extended based on the following criteria:

- Numeric constants are never sign extended in user mode. In kernel mode, a numeric constant is sign extended unless it contains a grave accent (` ) before the low bytes. For example, in kernel mode, the hexadecimal numbers **EEAA1122** and **00000000EEAA1122** are sign extended, but **00000000`EEAA1122** and **0`EEAA1122** are not.
- A 32-bit register is sign extended in both modes.
- Pseudo-registers are always stored as 64-bit values. They are not sign extended when they are evaluated. When a pseudo-register is *assigned* a value, the expression that is used is evaluated according to the standard C++ criteria.
- Individual numbers and registers in an expression can be sign extended, but no other calculations during expression evaluation are sign extended. As a result, you can mask the high bits of a number or register by using the following syntax.

```
( 0x0`FFFFFFFF & expression )
```

Sign Extension in C++ Expressions

When the debugger evaluates a C++ expression, the following rules apply:

- Registers and pseudo-registers are never sign extended.
- All other values are treated exactly like C++ would treat values of their type.

Displaying Sign-Extended and 64-Bit Numbers

Other than 32-bit and 16-bit registers, all numbers are stored internally within the debugger as 64-bit values. However, when a number satisfies certain criteria, the debugger displays it as a 32-bit number in command output.

The debugger uses the following criteria to determine how to display numbers:

- If the high 32 bits of a number are all zeros (that is, if the number is from 0x00000000`00000000 through 0x00000000`FFFFFFFF), the debugger displays the number as a 32-bit number.
- If the high 32 bits of a number are all ones and if the highest bit of the low 32 bits is also a one (that is, if the number is from 0xFFFFFFFF`80000000 through 0xFFFFFFFF`FFFFFFFF), the debugger assumes the number is a sign-extended 32-bit number and displays it as a 32-bit number.
- If the previous two condition do not apply (that is, if the number is from 0x00000001`00000000 through 0xFFFFFFFF`7FFFFFFF) the debugger displays the number as a 64-bit number.

Because of these display rules, when a number is displayed as a 32-bit number from 0x80000000 through 0xFFFFFFFF, you cannot confirm whether the high 32 bits are all ones or all zeros. To distinguish between these two cases, you must perform an additional computation on the number (such as masking one or more of the high bits and displaying the result).

String Wildcard Syntax

12/5/2018 • 2 minutes to read • Edit Online

Some debugger commands have string parameters that accept a variety of wildcard characters. These parameters are noted on their respective reference pages.

These kinds of parameters support the following syntax features:

- An asterisk (*) represents zero or more characters.
- A question mark (?) represents any single character.
- Brackets ([ ]) that contain a list of characters represent any single character in the list. Exactly one character in the list is matched. Within these brackets, you can use a hyphen (-) to specify a range. For example, **Prog[er-t7]am** matches "Progeam", "Program", "Progsam", "Progtam", and "Prog7am".
- A number sign (#) represents zero or more of the preceding characters. For example, **Lo#p** matches "Lp", "Lop", "Loop", "Looop", and so on. You can also combine a number sign with brackets, so **m[ia]#n** matches "mn", "min", "man", "maan", "main", "mian", "miin", "miain", and so on.
- A plus sign (+) represents one or more of the preceding characters. For example, **Lo+p** is the same as **Lo#p**, except that **Lo+p** does not match "Lp". Similarly, **m[ia]+n** is the same as **m[ia]#n**, except that m**[ia]+n** does not match "mn". **a?+b** is also the same as **a*b**, except that **a?+b** does not match "ab".
- If you have to specify a literal number sign (#), question mark (?), opening bracket ([), closing bracket (]), asterisk (*), or plus sign (+) character, you must add a backslash (\ ) in front of the character. Hyphens are always literal when you do not enclose them in brackets. But you cannot specify a literal hyphen within a bracketed list.

Parameters that specify symbols also support some additional features. In addition to the standard string wildcard characters, you can use an underscore (_) before a text expression that you use to specify a symbol. When matching this expression to a symbol, the debugger treats the underscore as any quantity of underscores, even zero. This feature applies only when you are matching symbols. It does not apply to string wildcard expressions in general. For more information about symbol syntax, see Symbol Syntax and Symbol Matching.

Register Syntax

12/5/2018 • 2 minutes to read • Edit Online

The debugger can control registers and floating-point registers.

When you use a register in an expression, you should add an at sign (**@**) before the register. This at sign tells the debugger that the following text is the name of a register.

If you are using MASM expression syntax, you can omit the at sign for certain very common registers. On x86-based systems, you can omit the at sign for the **eax**, **ebx**, **ecx**, **edx**, **esi**, **edi**, **ebp**, **eip**, and **efl** registers. However, if you specify a less common register without an at sign, the debugger first tries to interpret the text as a hexadecimal number. If the text contains non-hexadecimal characters, the debugger next interprets the text as a symbol. Finally, if the debugger does not find a symbol match, the debugger interprets the text as a register.

If you are using C++ expression syntax, the at sign is always required.

The **r (Registers)** command is an exception to this rule. The debugger always interprets its first argument as a register. (An at sign is not required or permitted.) If there is a second argument for the **r** command, it is interpreted according to the default expression syntax. If the default expression syntax is C++, you must use the following command to copy the **ebx** register to the **eax** register.

```
0:000> r eax = @ebx
```

For more information about the registers and instructions that are specific to each processor, see Processor Architecture.

Flags on an x86-based Processor

x86-based processors also use several 1-bit registers known as *flags*. For more information about these flags and the syntax that you can use to view or change them, see x86 Flags.

Registers and Threads

Each thread has its own register values. These values are stored in the CPU registers when the thread is executing and in memory when another thread is executing.

In user mode, any reference to a register is interpreted as the register that is associated with the current thread. For more information about the current thread, see Controlling Processes and Threads.

In kernel mode, any reference to a register is interpreted as the register that is associated with the current register context. You can set the register context to match a specific thread, context record, or trap frame. You can display only the most important registers for the specified register context, and you cannot change their values.

Pseudo-Register Syntax

3/6/2019 • 9 minutes to read • Edit Online

The debugger supports several pseudo-registers that hold certain values.

The debugger sets *automatic pseudo-registers* to certain useful values. *User-defined pseudo-registers* are integer variables that you can write to or read.

All pseudo-registers begin with a dollar sign (**$**). If you are using MASM syntax, you can add an at sign (**@**) before the dollar sign. This at sign tells the debugger that the following token is a register or pseudo-register, not a symbol. If you omit the at sign, the debugger responds more slowly, because it has to search the whole symbol table.

For example, the following two commands produce the same output, but the second command is faster.

```
0:000> ? $exp
Evaluate expression: 143 = 0000008f
0:000> ? @$exp
Evaluate expression: 143 = 0000008f
```

If a symbol exists with the same name as the pseudo-register, you must add the at sign.

If you are using C++ expression syntax, the at sign (**@**) is always required.

The **r (Registers)** command is an exception to this rule. The debugger always interprets its first argument as a register or pseudo-register. (An at sign is not required or permitted.) If there is a second argument for the **r** command, it is interpreted according to the default expression syntax. If the default expression syntax is C++, you must use the following command to copy the **$t2** pseudo-register to the **$t1** pseudo-register.

```
0:000> r $t1 = @$t2
```

Automatic Pseudo-Registers

The debugger automatically sets the following pseudo-registers.

PSEUDO-REGISTER	DESCRIPTION
$ea	The effective address of the last instruction that was executed. If this instruction does not have an effective address, the debugger displays "Bad register error". If this instruction has two effective addresses, the debugger displays the first address.
$ea2	The second effective address of the last instruction that was executed. If this instruction does not have two effective addresses, the debugger displays "Bad register error".
$exp	The last expression that was evaluated.

PSEUDO-REGISTER	DESCRIPTION
$ra	The return address that is currently on the stack. This address is especially useful in execution commands. For example, **g @$ra** continues until the return address is found (although **gu (Go Up)** is a more precise effective way of "stepping out" of the current function).
$ip	The instruction pointer register. **x86-based processors:** The same as **eip**. **Itanium-based processors:** Related to **iip**. (For more information, see the note following this table.) **x64-based processors:** The same as **rip**.
$eventip	The instruction pointer at the time of the current event. This pointer typically matches **$ip**, unless you switched threads or manually changed the value of the instruction pointer.
$previp	The instruction pointer at the time of the previous event. (Breaking into the debugger counts as an event.)
$relip	An instruction pointer that is related to the current event. When you are branch tracing, this pointer is the pointer to the branch source.
$scopeip	The instruction pointer for the current local context (also known as the *scope*).
$exentry	The address of the entry point of the first executable of the current process.
$retreg	The primary return value register. **x86-based processors:** The same as **eax**. **Itanium-based processors:** The same as **ret0**. **x64-based processors:** The same as **rax**.
$retreg64	The primary return value register, in 64-bit format. **x86 processor:** The same as the **edx:eax** pair.
$csp	The current call stack pointer. This pointer is the register that is most representative of call stack depth. **x86-based processors:** The same as **esp**. **Itanium-based processors:** The same as **bsp**. **x64-based processors:** The same as **rsp**.

PSEUDO-REGISTER	DESCRIPTION
$p	The value that the last **d* (Display Memory)** command printed.
$proc	The address of the current process (that is, the address of the EPROCESS block).
$thread	The address of the current thread. In kernel-mode debugging, this address is the address of the ETHREAD block. In user-mode debugging, this address is the address of the thread environment block (TEB).
$peb	The address of the process environment block (PEB) of the current process.
$teb	The address of the thread environment block (TEB) of the current thread.
$tpid	The process ID (PID) for the process that owns the current thread.
$tid	The thread ID for the current thread.
$dtid	
$dpid	
$dsid	
$bp*Number*	The address of the corresponding breakpoint. For example, **$bp3** (or **$bp03**) refers to the breakpoint whose breakpoint ID is 3. *Number* is always a decimal number. If no breakpoint has an ID of *Number*, **$bp***Number* evaluates to zero. For more information about breakpoints, see Using Breakpoints.
$frame	The current frame index. This index is the same frame number that the **.frame (Set Local Context)** command uses.
$dbgtime	The current time, according to the computer that the debugger is running on.

PSEUDO-REGISTER	DESCRIPTION
$callret	The return value of the last function that **.call (Call Function)** called or that is used in an **.fnret /s** command. The data type of **$callret** is the data type of this return value.
$extret	
$extin	
$clrex	
$lastclrex	**Managed debugging only:** The address of the last-encountered common language runtime (CLR) exception object.
$ptrsize	The size of a pointer. In kernel mode, this size is the pointer size on the target computer.
$pagesize	The number of bytes in one page of memory. In kernel mode, this size is the page size on the target computer.
$pcr	
$pcrb	
$argreg	
$exr_chance	The chance of the current exception record.
$exr_code	The exception code for the current exception record.
$exr_numparams	The number of parameters in the current exception record.
$exr_param0	The value of Parameter 0 in the current exception record.
$exr_param1	The value of Parameter 1 in the current exception record.
$exr_param2	The value of Parameter 2 in the current exception record.

PSEUDO-REGISTER	DESCRIPTION
$exr_param3	The value of Parameter 3 in the current exception record.
$exr_param4	The value of Parameter 4 in the current exception record.
$exr_param5	The value of Parameter 5 in the current exception record.
$exr_param6	The value of Parameter 6 in the current exception record.
$exr_param7	The value of Parameter 7 in the current exception record.
$exr_param8	The value of Parameter 8 in the current exception record.
$exr_param9	The value of Parameter 9 in the current exception record.
$exr_param10	The value of Parameter 10 in the current exception record.
$exr_param11	The value of Parameter 11 in the current exception record.
$exr_param12	The value of Parameter 12 in the current exception record.
$exr_param13	The value of Parameter 13 in the current exception record.
$exr_param14	The value of Parameter 14 in the current exception record.
$bug_code	If a bug check has occurred, this is the bug code. Applies to live kernel-mode debugging and kernel crash dumps.
$bug_param1	If a bug check has occurred, this is the value of Parameter 1. Applies to live kernel-mode debugging and kernel crash dumps.
$bug_param2	If a bug check has occurred, this is the value of Parameter 2. Applies to live kernel-mode debugging and kernel crash dumps.

PSEUDO-REGISTER	DESCRIPTION
$bug_param3	If a bug check has occurred, this is the value of Parameter 3. Applies to live kernel-mode debugging and kernel crash dumps.
$bug_param4	If a bug check has occurred, this is the value of Parameter 4. Applies to live kernel-mode debugging and kernel crash dumps.

Some of these pseudo-registers might not be available in certain debugging scenarios. For example, you cannot use **$peb**, **$tid**, and **$tpid** when you are debugging a user-mode minidump or certain kernel-mode dump files. There will be situations where you can learn thread information from **~ (Thread Status)** but not from **$tid**. You cannot use the **$previp** pseudo-register on the first debugger event. You cannot use the **$relip** pseudo-register unless you are branch tracing. If you use an unavailable pseudo-register, a syntax error occurs.

A pseudo-register that holds the address of a structure -- such as **$thread**, **$proc**, **$teb**, **$peb**, and **$lastclrex** -- will be evaluated according to the proper data type in the C++ expression evaluator, but not in the MASM expression evaluator. For example, the command **? $teb** displays the address of the TEB, while the command **?? @$teb** displays the entire TEB structure. For more information, see Evaluating Expressions.

On an Itanium-based processor, the **iip** register is *bundle-aligned*, which means that it points to slot 0 in the bundle containing the current instruction, even if a different slot is being executed. So **iip** is not the full instruction pointer. The **$ip** pseudo-register is the actual instruction pointer, including the bundle and the slot. The other pseudo-registers that hold address pointers (**$ra**, **$retreg**, **$eventip**, **$previp**, **$relip**, and **$exentry**) have the same structure as **$ip** on all processors.

You can use the **r** command to change the value of **$ip**. This change also automatically changes the corresponding register. When execution resumes, it resumes at the new instruction pointer address. This register is the only automatic pseudo-register that you can change manually.

Note In MASM syntax, you can indicate the **$ip** pseudo-register with a period (**.**). You do not add an at sign (@) before this period, and do not use the period as the first parameter of the **r** command. This syntax is not permitted within a C++ expression.

Automatic pseudo-registers are similar to automatic aliases. But you can use automatic aliases together with alias-related tokens (such as **${ }**), and you cannot use pseudo-registers with such tokens.

User-Defined Pseudo-Registers

There are 20 user-defined pseudo-registers (**$t0**, **$t1**, ..., **$t19**). These pseudo-register are variables that you can read and write through the debugger. You can store any integer value in these pseudo-registers. They can be especially useful as loop variables.

To write to one of these pseudo-registers, use the **r (Registers)** command, as the following example shows.

```
0:000> r $t0 = 7
0:000> r $t1 = 128*poi(MyVar)
```

Like all pseudo-registers, you can use the user-defined pseudo-register in any expression, as the following example shows.

```
0:000> bp $t3
0:000> bp @$t4
0:000> ?? @$t1 + 4*@$t2
```

A pseudo-register is always typed as an integer, unless you use the **?** switch together with the **r** command. If you use this switch, the pseudo-register acquires the type of whatever is assigned to it. For example, the following command assigns the UNICODE_STRING** type and the 0x0012FFBC value to **$t15**.

```
0:000> r? $t15 = * (UNICODE_STRING*) 0x12ffbc
```

User-defined pseudo-registers use zero as the default value when the debugger is started.

Note The aliases **$u0**, **$u1**, ..., **$u9** are not pseudo-registers, despite their similar appearance. For more information about these aliases, see Using Aliases.

Example

The following example sets a breakpoint that is hit every time that the current thread calls **NtOpenFile**. But this breakpoint is not hit when other threads call **NtOpenFile**.

```
kd> bp /t @$thread nt!ntopenfile
```

Example

The following example executes a command until the register holds a specified value. First, put the following code for conditional stepping in a script file named "eaxstep".

```
.if (@eax == 1234) { .echo 1234 } .else { t "$<eaxstep" }
```

Next, issue the following command.

```
t "$<eaxstep"
```

The debugger performs a step and then runs your command. In this case, the debugger runs the script, which either displays **1234** or repeats the process.

Source Line Syntax

12/5/2018 • 2 minutes to read • Edit Online

You can specify source file line numbers as all or part of an MASM expression. These numbers evaluate to the offset of the executable code that corresponds to this source line.

Note You cannot use source line numbers as part of a C++ expression. For more information about when MASM and C++ expression syntax is used, see Evaluating Expressions.

You must enclose source file and line number expressions by grave accents (`). The following example shows the full format for source file line numbers.

```
`[[Module!]Filename][:LineNumber]`
```

If you have multiple files that have identical file names, *Filename* should include the whole directory path and file name. This directory path should be the one that is used at compilation time. If you supply only the file name or only part of the path and if there are multiple matches, the debugger uses the first match that it finds.

If you omit *Filename*, the debugger uses the source file that corresponds to the current program counter.

LineNumber is read as a decimal number unless you precede it with **0x**, regardless of the current default radix. If you omit *LineNumber*, the expression evaluates to the initial address of the executable that corresponds to the source file.

Source line expressions are not evaluated in CDB unless you issue a **.lines (Toggle Source Line Support)** command or you include the **-lines command-line option** when you start WinDbg..

For more information about source debugging, see Debugging in Source Mode.

Address and Address Range Syntax

6/17/2019 • 4 minutes to read • Edit Online

There are several ways to specify addresses in the debugger.

Addresses are always *virtual addresses*, except when the documentation specifically indicates another kind of address. In user mode, the debugger interprets virtual addresses according to the page directory of the current process. In kernel mode, the debugger interprets virtual addresses according to the page directory of the process that the process context specifies. You can also directly set the *user-mode address context*. For more information about the user-mode address context, see **.context (Set User-Mode Address Context)**.

Address Modes and Segment Support

On x86-based platforms, CDB and KD support the following addressing modes. These modes are distinguished by their prefixes.

PREFIX	NAME	ADDRESS TYPES
%	flat	32-bit addresses (also 16-bit selectors that point to 32-bit segments) and 64-bit addresses on 64-bit systems.
&	virtual 86	Real-mode addresses. x86-based only.
#	plain	Real-mode addresses. x86-based only.

The difference between the plain and virtual 86 modes is that a plain 16-bit address uses the segment value as a selector and looks up the segment descriptor. But a virtual 86 address does not use selectors and instead maps directly into the lower 1 MB.

If you access memory through an addressing mode that is not the current default mode, you can use the address mode prefixes to override the current address mode.

Address Arguments

Address arguments specify the location of variables and functions. The following table explains the syntax and meaning of the various addresses that you can use in CDB and KD.

SYNTAX	MEANING
offset	The absolute address in virtual memory space, with a type that corresponds to the current execution mode. For example, if the current execution mode is 16 bit, the offset is 16 bit. If the execution mode is 32-bit segmented, the offset is 32-bit segmented.
&[[segment:]] offset	The real address. x86-based and x64-based.

SYNTAX	MEANING
%segment:[[offset]]	A segmented 32-bit or 64-bit address. x86-based and x64-based.
%[[offset]]	An absolute address (32-bit or 64-bit) in virtual memory space. x86-based and x64-based.
name[[**+|−**]] offset	A flat 32-bit or 64-bit address. *name* can be any symbol. *offset* specifies the offset. This offset can be whatever address mode its prefix indicates. No prefix specifies a default mode address. You can specify the offset as a positive (+) or negative (−) value.

Use the **dg (Display Selector)** command to view segment descriptor information.

In MASM expressions, you can also use the **poi** operator to dereference any pointer. For example, if the pointer at address 0x00123456 points to address location 0x00420000, the following two commands are equivalent.

```
0:000> dd 420000
0:000> dd poi(123456)
```

In C++ expressions, pointers behave like pointers in C++. However, numbers are interpreted as integers. If you have to deference an actual number, you must cast it first, as the following example shows.

```
0:000> dd *( (long*) 0x123456 )
```

Some pseudo-registers also hold common addresses, such as the current program counter location.

You can also indicate an address in an application by specifying the original source file name and line number. For more information about how to specify this information, see Source Line Syntax.

Address Ranges

You can specify an address range by a pair of addresses or by an address and object count.

To specify a range by a pair of addresses, specify the starting address and the ending address. For example, the following example is a range of 8 bytes, beginning at the address 0x00001000.

```
0x00001000  0x00001007
```

To specify an address range by an address and object count, specify an address argument, the letter L (uppercase or lowercase), and a value argument. The address specifies the starting address. The value specifies the number of objects to be examined or displayed. The size of the object depends on the command. For example, if the object size is 1 byte, the following example is a range of 8 bytes, beginning at the address 0x00001000.

```
0x00001000  L8
```

However, if the object size is a double word (32 bits or 4 bytes), the following two ranges each give an 8-byte range.

```
0x00001000  0x00001007
0x00001000  L2
```

There are two other ways to specify the value (the **L***Size* range specifier):

- **L?** *Size* (with a question mark) means the same as **L***Size*, except that **L?** *Size* removes the debugger's automatic range limit. Typically, there is a range limit of 256 MB, because larger ranges are typographic errors. If you want to specify a range that is larger than 256 MB, you must use the **L?** *Size* syntax.
- **L-** *Size* (with a hyphen) specifies a range of length *Size* that ends at the given address. For example, **80000000 L20** specifies the range from 0x80000000 through 0x8000001F, and **80000000 L-20** specifies the range from 0x7FFFFFE0 through 0x7FFFFFFF.

Some commands that ask for address ranges accept a single address as the argument. In this situation, the command uses some default object count to compute the size of the range. Typically, commands for which the address range is the final parameter permit this syntax. For the exact syntax and the default range size for each command, see the reference topics for each command.

Thread Syntax

12/5/2018 • 2 minutes to read • Edit Online

Many debugger commands have thread identifiers as their parameters. A tilde (~) appears before the thread identifier.

The thread identifier can be one of the following values.

THREAD IDENTIFIER	DESCRIPTION
~.	The current thread.
~#	The thread that caused the current exception or debug event.
~*	All threads in the process.
~*Number*	The thread whose index is *Number*.
~~[*TID*]	The thread whose thread ID is *TID*. (The brackets are required And you cannot add a space between the second tilde and the opening bracket.)
~[*Expression*]	The thread whose thread ID is the integer to which the numerical *Expression* resolves.

Threads are assigned indexes as they are created. Note that this number differs from the thread ID that the Microsoft Windows operating system uses.

When debugging begins, the current thread is the one that caused the present exception or debug event (or the active thread when the debugger attached to the process). That thread remains the current thread until you specify a new one by using a **~s (Set Current Thread)** command or by using the Processes and Threads window in WinDbg.

Thread identifiers typically appear as command prefixes. Note that not all wildcard characters are available in all commands that use thread identifiers.

An example of the ~[*Expression*] syntax would be `~[@$t0]`. In this example, the thread changes depending on the value of a user-defined pseudo-register. This syntax allows debugger scripts to programmatically select a thread.

Controlling Threads in Kernel Mode

In kernel mode, you cannot control threads by using thread identifiers. For more information about how to access thread-specific information in kernel mode, see Changing Contexts.

Note You can use the tilde character (~) to specify threads during user-mode debugging. In kernel-mode debugging, you can use the tilde to specify processors. For more information about how to specify processors, see Multiprocessor Syntax.

Process Syntax

10/25/2019 • 2 minutes to read • Edit Online

Many debugger commands have process identifiers as their parameters. A vertical bar (|) appears before the process identifier.

The process identifier can be one of the following values.

PROCESS IDENTIFIER	DESCRIPTION
|.	The current process.
|#	The process that caused the current exception or debug event.
|*	All processes.
|*Number*	The process whose ordinal is *Number*.
|~[*PID***]**	The process whose process ID is *PID*. (The brackets are required and you cannot add a space between the tilde (~) and the opening bracket.)
|[*Expression***]**	The process whose process ID is the integer to which the numerical *Expression* resolves.

Processes are assigned ordinals as they are created. Note that this number differs from the process ID (PID) that the Microsoft Windows operating system uses.

The current process defines the memory space and the set of threads that are used. When debugging begins, the current process is the one that caused the present exception or debug event (or the process that the debugger attached to). That process remains the current process until you specify a new one by using a **|s (Set Current Process)** command or by using the Processes and Threads window in WinDbg.

Process identifiers are used as parameters in several commands, frequently as the command prefix. Note that WinDbg and CDB can debug child processes that the original process created. WinDbg and CDB can also attach to multiple unrelated processes.

An example of the |[*Expression*] syntax would be |[@$t0]. In this example, the process changes depending on the value of a user-defined pseudo-register. This syntax allows debugger scripts to programmatically select a process.

Controlling Processes in Kernel Mode

In kernel mode, you cannot control processes by using process identifiers. For more information about how to access process-specific information in kernel mode, see Changing Contexts.

System Syntax

12/5/2018 • 2 minutes to read • Edit Online

Many debugger commands have process identifiers as their parameters.

Two vertical bars (||) appear before the system identifier. The system identifier can be one of the following values.

SYSTEM IDENTIFIER	DESCRIPTION
||.	The current system
||#	The system that caused the current exception or debug event.
||*	All systems.
||*ddd*	The system whose ordinal is *ddd*.

Systems are assigned ordinals in the order that the debugger attaches to them.

When debugging begins, the current system is the one that caused the present exception or debug event (or the one that the debugger most recently attached to). That system remains the current system until you specify a new one by using a **||s (Set Current System)** command or by using the Processes and Threads window in WinDbg.

Example

This example shows three dump files are loaded. System 1 is active and system 2 caused the debug event.

```
||1:1:017> ||
   0 User mini dump: c:\notepad.dmp
.  1 User mini dump: c:\paint.dmp
#  2 User mini dump: c:\calc.dmp
```

Remarks

To work with multiple systems, you can use the .opendump to debug multiple crash dumps at the same time. For more information about how to control a multiple-target session, see Debugging Multiple Targets.

Note There are complications, when you debug live targets and dump targets together, because commands behave differently for each type of debugging. For example, if you use the **g (Go)** command when the current system is a dump file, the debugger begins executing, but you cannot break back into the debugger, because the break command is not recognized as valid for dump file debugging.

Multiprocessor Syntax

12/5/2018 • 2 minutes to read • Edit Online

KD and kernel-mode WinDbg support multiple processor debugging. You can perform this kind of debugging on any multiprocessor platform.

Processors are numbered zero through *n*.

If the current processor is processor 0 (that is, if it is the processor that currently caused the debugger to be active), you can examine the other non-current processors (processors one through *n*). However, you cannot change anything in the non-current processors. You can only view their state.

Selecting a Processor

You can use the **.echocpunum (Show CPU Number)** command to display the processor numbers of the current processor. The output from this command enables you to immediately tell when you are working on a multiple processor system by the text in the kernel debugging prompt.

In the following example, **0:** in front of the **kd>** prompt indicates that you are debugging the first processor in the computer.

```
0: kd>
```

Use the **~s (Change Current Processor)** command to switch between processors, as the following example shows.

```
0: kd> ~1s
1: kd>
```

Now you are debugging the second processor in the computer.

You might have to change processors on a multiprocessor system if you encounter a break and you cannot understand the stack trace. The break might have occurred on a different processor.

Specifying Processors in Other Commands

You can add a processor number before several commands. This number is not preceded by a tilde (**~**), except in the **~S** command.

Note In user-mode debugging, the tilde is used to specify threads. For more information about this syntax, see Thread Syntax.

Processor IDs do not have to be referred to explicitly. Instead, you can use a numerical expression that resolves to an integer that corresponds to a processor ID. To indicate that the expression should be interpreted as a processor, use the following syntax.

```
||[Expression]
```

In this syntax, the square brackets are required, and *Expression* stands for any numerical expression that resolves to an integer that corresponds to a processor ID.

In the following example, the processor changes depending on the value of a user-defined pseudo-register.

```
||[@$t0]
```

Examples

The following example uses the **k (Display Stack Backtrace)** command to display a stack trace from processor two.

```
1: kd> 2k
```

The following example uses the **r (Registers)** command to display the **eax** register of processor three.

```
1: kd> 3r eax
```

However, the following command gives a syntax error, because you cannot change the state of a processor other than the current processor.

```
1: kd> 3r eax=808080
```

Breakpoints

During kernel debugging, the **bp, bu, bm (Set Breakpoint)** and **ba (Break on Access)** commands apply to all processors of a multiple processor computer.

For example, if the current processor is three, you can enter the following command to put a breakpoint at **SomeAddress**.

```
1: kd> bp SomeAddress
```

Then, any processor (not only processor one) that executes at that address causes a breakpoint trap.

Displaying Processor Information

You can use the **!running** extension to display the status of each processor on the target computer. For each processor, **!running** can also display the current and next thread fields from the process control block (PRCB), the state of the 16 built-in queued spinlocks, and a stack trace.

You can use the **!cpuinfo** and **!cpuid** extensions to display information about the processors themselves.

Command Tokens

6/25/2019 • 2 minutes to read • Edit Online

This section of the reference discusses the various tokens used within debugger commands and meta-commands.

These tokens include:

; (Command Separator)

{ } (Block Delimiter)

${ } (Alias Interpreter)

$$ (Comment Specifier)

*** (Comment Line Specifier)**

.block

.break

.catch

.continue

.do

.else

.elsif

.for

.foreach

.if

.leave

.printf

.while

; (Command Separator)

12/5/2018 • 2 minutes to read • Edit Online

The semicolon (**;**) character is used to separate multiple commands on a single line.

```
Command1 ; Command2 [; Command3 ...]
```

Parameters

Command1, *Command2*, ...
The commands to be executed.

Remarks

Commands are executed sequentially from left to right. All commands on a single line refer to the current thread, unless otherwise specified. If a command causes the thread to execute, the remaining commands on the line will be deferred until that thread stops on a debug event.

A small number of commands cannot be followed by a semicolon, because they automatically take the entire remainder of the line as their argument. These include **as (Set Alias)**, **$< (Run Script File)**, **$>< (Run Script File)**, and any command beginning with the *** (Comment Line Specifier)** token.

Here is an example. This executes the current program to source line 123, prints the value of **counter**, then resumes execution:

```
0:000> g `:123`; ? poi(counter); g
```

{ } (Block Delimiter)

12/5/2018 • 2 minutes to read • Edit Online

A pair of braces (**{ }**) is used to surround a block of statements within a debugger command program.

```
Statements { Statements } Statements
```

Additional Information

For information about debugger command programs and control flow tokens, see Using Debugger Command Programs.

Remarks

When each block is entered, all aliases within the block are evaluated. If you alter the value of an alias at some point within a command block, commands subsequent to that point will not use the new alias value unless they are within a subordinate block.

Each block must begin with a control flow token. If you wish to create a block for the sole purpose of evaluating aliases, you should prefix it with the **.block** token.

${ } (Alias Interpreter)

12/5/2018 • 2 minutes to read • Edit Online

A dollar sign followed by a pair of braces (**${ }**) evaluates to a variety of values related to the specified user-named alias.

```
Text ${Alias} Text
Text ${/d:Alias} Text
Text ${/f:Alias} Text
Text ${/n:Alias} Text
Text ${/v:Alias} Text
```

Parameters

Alias
Specifies the name of the alias to be expanded or evaluated. *Alias* must be a user-named alias or the *Variable* value used by the **.foreach** token.

/d
Evaluates to one or zero depending on whether the alias is currently defined. If the alias is defined, **${/v:***Alias***}** is replaced by 1; if the alias is not defined, **${/v:***Alias***}** is replaced by 0.

/f
Evaluates to the alias equivalent if the alias is currently defined. If the alias is defined, **${/f:***Alias***}** is replaced by the alias equivalent; if the alias is not defined, **${/f:***Alias***}** is replaced by an empty string.

/n
Evaluates to the alias name if the alias is currently defined. If the alias is defined, **${/n:***Alias***}** is replaced by the alias name; if the alias is not defined, **${/n:***Alias***}** is not replaced but retains its literal value of **${/n:***Alias***}**.

/v
Prevents any alias evaluation. Regardless of whether *Alias* is defined, **${/v:***Alias***}** always retains its literal value of **${/v:***Alias***}**.

Additional Information

For an explanation of how to use aliases, see Using Aliases.

Remarks

If no switches are used and the alias is currently defined, **${***Alias***}** is replaced by the alias equivalent. If no switches are used and the alias is not defined, **${***Alias***}** always retains its literal value of **${***Alias***}**.

One advantage of using the **${ }** token is that the alias will be evaluated even if it is adjacent to other characters. Without this token, the debugger only replaces aliases that are separated from other tokens by a space.

As indicated, there are circumstances where the **${ }** token is not replaced by anything but retains its literal value. This occurs when no switch is used and *Alias* is undefined, when the **/n** switch is used and *Alias* is undefined, and always when the **/v** switch is used. In these circumstances, the token retains its literal value, including the dollar sign and the braces. Therefore, if this is used as the parameter of a command, a syntax error will result, unless that parameter accepts arbitrary text strings.

There is, however, one exception to this. If you use **${/v:***Alias***}** as the first parameter to the **as (Set Alias)** or **aS (Set Alias)** command, this token will be treated as the string *Alias* alone, not as the string **${/v:***Alias***}**. This only works with the **as**, **aS**, and **ad** commands, and it only works when the **/v** switch is used—it will not work with **${/n:***Alias***}** or **${***Alias***}** when they retain their literal values.

Alias must be a user-named alias or the *Variable* value used by the **.foreach** token—not a fixed-name alias. If there is a fixed-name alias within the string *Alias*, it will be replaced before the **${ }** token is evaluated.

$$ (Comment Specifier)

10/8/2019 • 2 minutes to read • Edit Online

If two dollar signs (**$$**) appear at the start of a command, then the rest of the line is treated as a comment, unless the comment is terminated by a semicolon.

$$ [any text]

Remarks

The **$$** token is parsed like any other debugger command. Therefore, if you want to create a comment after another command, you must precede the **$$** token with a semicolon.

The **$$** token will cause the text after it to be ignored until the end of the line or until a semicolon is encountered. A semicolon terminates the comment; text after the semicolon is parsed as a standard command. This differs from the ***** (Comment Line Specifier) which makes the remainder of the line a comment even if a semicolon is present.

For example, the following command will display **eax** and **ebx**, but not **ecx**:

```
0:000> r eax; $$ some text; r ebx; * more text; r ecx
```

Text prefixed by the ***** or **$$** tokens is not processed in any way. If you are performing remote debugging, a comment entered in the debugging server will not be visible in the debugging client, nor vice-versa. If you wish to make comment text appear in the Debugger Command window in a way visible to all parties, you should use **.echo (Echo Comment)**.

* (Comment Line Specifier)

10/8/2019 • 2 minutes to read • Edit Online

If the asterisk (*****) character is at the start of a command, then the rest of the line is treated as a comment, even if a semicolon appears after it.

```
* [any text]
```

Remarks

The ***** token is parsed like any other debugger command. Therefore, if you want to create a comment after another command, you must precede the ***** token with a semicolon.

The ***** token will cause the remainder of the line to be ignored, even if a semicolon appears after it. This differs from **$$ (Comment Specifier)**, which creates a comment that can be terminated by a semicolon.

For example, the following command will display **eax** and **ebx**, but not **ecx**:

```
0:000> r eax; $$ some text; r ebx; * more text; r ecx
```

Text prefixed by the ***** or **$$** tokens is not processed in any way. If you are performing remote debugging, a comment entered in the debugging server will not be visible in the debugging client, nor vice-versa. If you wish to make comment text appear in the Debugger Command window in a way visible to all parties, you should use **.echo (Echo Comment)**.

.block

12/5/2018 • 2 minutes to read • Edit Online

The **.block** token performs no action; it is used solely to introduce a block of statements.

```
Commands ; .block { Commands } ; Commands
```

Additional Information

For information about using a new block to evaluate an alias, see Using Aliases and **as, aS (Set Alias)**.

For information about other control flow tokens and their use in debugger command programs, see Using Debugger Command Programs.

Remarks

Blocks of commands are surrounded by braces. When each block is entered, all aliases within the block are evaluated. If you alter the value of an alias at some point within a command block, commands subsequent to that point will not use the new alias value unless they are within a subordinate block.

Each block must begin with a control flow token. If you wish to create a block for the sole purpose of evaluating aliases, you should prefix it with the **.block** token, since this token has no effect other than to allow a block to be introduced.

.break

12/5/2018 • 2 minutes to read • Edit Online

The **.break** token behaves like the **break** keyword in C.

```
 .for (...) { ... ; .if (Condition) .break ; ...}

.while (...) { ... ; .if (Condition) .break ; ...}

.do { ... ; .if (Condition) .break ; ...} (...)
```

Additional Information

For information about other control flow tokens and their use in debugger command programs, see Using Debugger Command Programs.

Remarks

The **.break** token can be used within any **.for**, **.while**, or **.do** loop.

Since there is no control flow token equivalent to the C **goto** statement, you will usually use the **.break** token within an **.if** conditional, as shown in the syntax examples above. However, this is not actually required.

.catch

12/5/2018 • 2 minutes to read • Edit Online

The **.catch** token is used to prevent a program from terminating if an error occurs.

It does not behave like the **catch** keyword in C++.

```
Commands ; .catch { Commands } ; Commands
```

Additional Information

For information about other control flow tokens and their use in debugger command programs, see Using Debugger Command Programs.

Remarks

The **.catch** token is followed by braces enclosing one or more commands.

If a command within a **.catch** block generates an error, the error message is displayed, all remaining commands within the braces are ignored, and execution resumes with the first command after the closing brace.

If **.catch** is not used, an error will terminate the entire debugger command program.

You can use **.leave** to exit from a **.catch** block.

.continue

12/5/2018 • 2 minutes to read • Edit Online

The **.continue** token behaves like the **continue** keyword in C.

```
.for (...) { ... ; .if (Condition) .continue ; ... }

.while (...) { ... ; .if (Condition) .continue ; ... }

.do { ... ; .if (Condition) .continue ; ... } (...)
```

Additional Information

For information about other control flow tokens and their use in debugger command programs, see Using Debugger Command Programs.

Remarks

The **.continue** token can be used within any **.for**, **.while**, or **.do** loop.

Since there is no control flow token equivalent to the C goto statement, you will usually use the **.continue** token within an **.if** conditional, as shown in the syntax examples above. However, this is not actually required.

.do

6/25/2019 • 2 minutes to read • Edit Online

The **.do** token behaves like the **do** keyword in C, except that the word "while" is not used before the condition.

```
.do { Commands } (Condition)
```

Syntax Elements

Commands
Specifies one or more commands that will be executed repeatedly as long as the condition is true -- but will always be executed at least once. This block of commands needs to be enclosed in braces, even if it consists of a single command. Multiple commands should be separated by semicolons, but the final command before the closing brace does not need to be followed by a semicolon.

Condition
Specifies a condition. If this evaluates to zero, it is treated as false; otherwise it is true. Enclosing *Condition* in parentheses is optional. *Condition* must be an expression, not a debugger command. It will be evaluated by the default expression evaluator (MASM or C++). For details, see Numerical Expression Syntax.

Additional Information

For information about other control flow tokens and their use in debugger command programs, see Using Debugger Command Programs.

Remarks

The **.break** and **.continue** tokens can be used to exit or restart the *Commands* block.

.else

12/5/2018 • 2 minutes to read • Edit Online

The **.else** token behaves like the **else** keyword in C.

```
.if (Condition) { Commands } .else { Commands }

.if (Condition) { Commands } .elsif (Condition) { Commands } .else { Commands }
```

Syntax Elements

Commands
Specifies one or more commands that will be executed conditionally. This block of commands needs to be enclosed in braces, even if it consists of a single command. Multiple commands should be separated by semicolons, but the final command before the closing brace does not need to be followed by a semicolon.

Additional Information

For information about other control flow tokens and their use in debugger command programs, see Using Debugger Command Programs.

.elsif

12/5/2018 • 2 minutes to read • Edit Online

The **.elsif** token behaves like the **else if** keyword combination in C.

```
.if (Condition) { Commands } .elsif (Condition) { Commands }

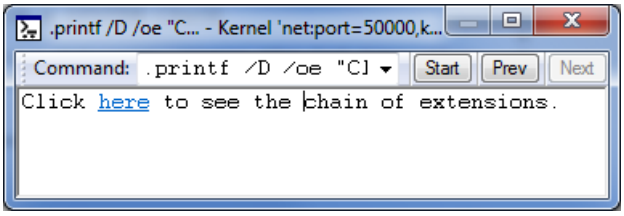
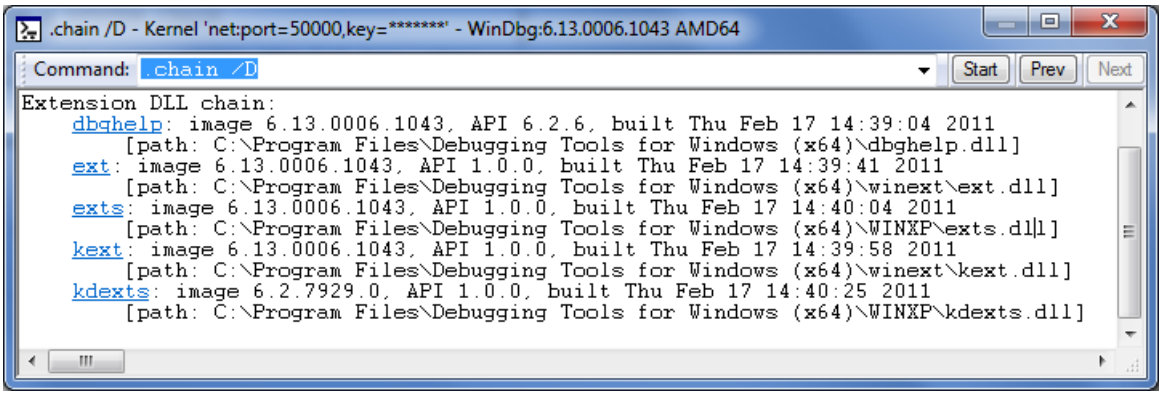
.if (Condition) { Commands } .elsif (Condition) { Commands } .else { Commands }
```

Syntax Elements

Condition
Specifies a condition. If this evaluates to zero, it is treated as false; otherwise it is true. Enclosing *Condition* in parentheses is optional. *Condition* must be an expression, not a debugger command. It will be evaluated by the default expression evaluator (MASM or C++). For details, see Numerical Expression Syntax.

Commands
Specifies one or more commands that will be executed conditionally. This block of commands needs to be enclosed in braces, even if it consists of a single command. Multiple commands should be separated by semicolons, but the final command before the closing brace does not need to be followed by a semicolon.

Additional Information

For information about other control flow tokens and their use in debugger command programs, see Using Debugger Command Programs.

.for

6/25/2019 • 2 minutes to read • Edit Online

The **.for** token behaves like the **for** keyword in C, except that multiple increment commands must be separated by semicolons, not by commas.

```
.for (InitialCommand ; Condition ; IncrementCommands) { Commands }
```

Syntax Elements

InitialCommand
Specifies a command that will be executed before the loop begins. Only a single initial command is permitted.

Condition
Specifies a condition. If this evaluates to zero, it is treated as false; otherwise it is true. Enclosing *Condition* in parentheses is optional. *Condition* must be an expression, not a debugger command. It will be evaluated by the default expression evaluator (MASM or C++). For details, see Numerical Expression Syntax.

IncrementCommands
Specifies one or more commands that will be executed at the conclusion of each loop. If you wish to use multiple increment commands, separate them by semicolons but do not enclose them in braces.

Commands
Specifies one or more commands that will be executed repeatedly as long as the condition is true. This block of commands needs to be enclosed in braces, even if it consists of a single command. Multiple commands should be separated by semicolons, but the final command before the closing brace does not need to be followed by a semicolon.

Additional Information

For information about other control flow tokens and their use in debugger command programs, see Using Debugger Command Programs.

Remarks

If all the work is being done by the increment commands, you can omit *Condition* entirely and simply use an empty pair of braces.

Here is an example of a **.for** statement with multiple increment commands:

```
0:000> .for (r eax=0; @eax < 7; r eax=@eax+1; r ebx=@ebx+1) { .... }
```

The **.break** and **.continue** tokens can be used to exit or restart the *Commands* block.

.foreach

12/5/2018 • 2 minutes to read • Edit Online

The **.foreach** token parses the output of one or more debugger commands and uses each value in this output as the input to one or more additional commands.

```
.foreach [Options] ( Variable  { InCommands } ) { OutCommands }

.foreach [Options] /s ( Variable  "InString" ) { OutCommands }

.foreach [Options] /f ( Variable  "InFile" ) { OutCommands }
```

Syntax Elements

Options
Can be any combination of the following options:

/pS *InitialSkipNumber*
Causes some initial tokens to be skipped. *InitialSkipNumber* specifies the number of output tokens that will not be passed to the specified *OutCommands*.

/ps *SkipNumber*
Causes tokens to be skipped repeatedly each time a command is processed. After each time a token is passed to the specified *OutCommands*, a number of tokens equal to the value of *SkipNumber* will be ignored.

Variable
Specifies a variable name. This variable will be used to hold the output from each command in the *InCommands* string; you can reference *Variable* by name in the parameters passed to the *OutCommands*. Any alphanumeric string can be used, although using a string that can also pass for a valid hexadecimal number or debugger command is not recommended. If the name used for *Variable* happens to match an existing global variable, local variable, or alias, their values will not be affected by the **.foreach** command.

InCommands
Specifies one or more commands whose output will be parsed; the resulting tokens will be passed to *OutCommands*. The output from *InCommands* is not displayed.

InString
Used with **/s**. Specifies a string that will be parsed; the resulting tokens will be passed to *OutCommands*.

InFile
Used with **/f**. Specifies a text file that will be parsed; the resulting tokens will be passed to *OutCommands*. The file name *InFile* must be enclosed in quotation marks.

OutCommands
Specifies one or more commands which will be executed for each token. Whenever the *Variable* string occurs it will be replaced by the current token.

Note When the string *Variable* appears within *OutCommands*, it must be surrounded by spaces. If it is adjacent to any other text -- even a parenthesis -- it will not be replaced by the current token value, unless you use the **${ } (Alias Interpreter)** token.

Additional Information

For information about other control flow tokens and their use in debugger command programs, see Using Debugger Command Programs.

Remarks

When the output from *InCommands*, the *InString* string, or the *InFile* file is parsed, any number of spaces, tabs, or carriage returns is treated as a single delimiter. Each of the resulting pieces of text is used to replace *Variable* when it appears within *OutCommands*.

Here is an example of a **.foreach** statement that uses the **dds** command on each token found in the file *myfile.txt*:

```
0:000> .foreach /f ( place "g:\myfile.txt") { dds place }
```

The **/pS** and **/ps** flags can be used to pass only certain tokens to the specified *OutCommands*. For example, the following statement will skip the first two tokens in the *myfile.txt* file and then pass the third to **dds**. After each token that is passed, it will skip four tokens. The result is that **dds** will be used with the 3rd, 8th, 13th, 18th, and 23rd tokens, and so on:

```
0:000> .foreach /pS 2 /ps 4 /f ( place "g:\myfile.txt") { dds place }
```

For more examples that use the **.foreach** token, see Debugger Command Program Examples.

.if

12/5/2018 • 2 minutes to read • Edit Online

The **.if** token behaves like the **if** keyword in C.

```
.if (Condition) { Commands }

.if (Condition) { Commands } .else { Commands }

.if (Condition) { Commands } .elsif (Condition) { Commands }

.if (Condition) { Commands } .elsif (Condition) { Commands } .else { Commands }
```

Syntax Elements

Condition
Specifies a condition. If this evaluates to zero, it is treated as false; otherwise it is true. Enclosing *Condition* in parentheses is optional. *Condition* must be an expression, not a debugger command. It will be evaluated by the default expression evaluator (MASM or C++). For details, see Numerical Expression Syntax.

Commands
Specifies one or more commands that will be executed conditionally. This block of commands needs to be enclosed in braces, even if it consists of a single command. Multiple commands should be separated by semicolons, but the final command before the closing brace does not need to be followed by a semicolon.

Additional Information

For information about other control flow tokens and their use in debugger command programs, see Using Debugger Command Programs.

.leave

12/5/2018 • 2 minutes to read • Edit Online

The **.leave** token is used to exit from a **.catch** block.

```
.catch { ... ; .if (Condition) .leave ; ... }
```

Additional Information

For information about other control flow tokens and their use in debugger command programs, see Using Debugger Command Programs.

Remarks

When a **.leave** token is encountered within a **.catch** block, the program exits from the block, and execution resumes with the first command after the closing brace.

Since there is no control flow token equivalent to the C **goto** statement, you will usually use the **.leave** token within an **.if** conditional, as shown in the syntax examples above. However, this is not actually required.

.printf

3/5/2019 • 3 minutes to read • Edit Online

The **.printf** token behaves like the **printf** statement in C.

```
.printf [/D] [Option] "FormatString" [, Argument , ...]
```

Syntax Elements

/D
Specifies that the format string contains Debugger Markup Language (DML).

Option
(WinDbg only) Specifies the type of text message that WinDbg should interpret the FormatString as. WinDbg assigns each type of Debugger Command window message a background and text color; choosing one of these options causes the message to be displayed in the appropriate colors. The default is to display the text as a normal-level message. For more information on message colors and how to set them, see View | Options.

The following options are available.

OPTION	TYPE OF MESSAGE	TITLE OF COLORS IN OPTIONS DIALOG BOX
/od	debuggee	Debuggee level command window
/oD	debuggee prompt	Debuggee prompt level command window
/oe	error	Error level command window
/on	normal	Normal level command window
/op	prompt	Prompt level command window
/oP	prompt registers	Prompt registers level command window
/os	symbols	Symbol message level command window
/ov	verbose	Verbose level command window
/ow	warning	Warning level command window

FormatString

Specifies the format string, as in **printf**. In general, conversion characters work exactly as in C. For the floating-point conversion characters, the 64-bit argument is interpreted as a 32-bit floating-point number unless the **l** modifier is used.

The "I64" modifier can be added to indicate that a value should be interpreted as 64-bits. For instance, "%I64x" can be used to print a 64-bit hexadecimal number.

The %p conversion character is supported, but it represents a pointer in the target's virtual address space. It must not have any modifiers and it uses the debugger's internal address formatting. In addition to the standard printf-style format specifiers, the following additional conversion characters are supported.

CHARACTER	ARGUMENT TYPE	ARGUMENT	TEXT PRINTED
%p	ULONG64	A pointer in the target's virtual address space.	The value of the pointer.
%N	DWORD_PTR (32 or 64 bits, depending on the host's architecture)	A pointer in the host's virtual address space.	The value of the pointer. (This is equivalent to the standard C %p character.)
%ma	ULONG64	The address of a NULL-terminated ASCII string in the target's virtual address space.	The specified string.
%mu	ULONG64	The address of a NULL-terminated Unicode string in the target's virtual address space.	The specified string.
%msa	ULONG64	The address of an ANSI_STRING structure in the target's virtual address space.	The specified string.
%msu	ULONG64	The address of a UNICODE_STRING structure in the target's virtual address space.	The specified string.
%y	ULONG64	The address of a debugger symbol in the target's virtual address space.	A string containing the name of the specified symbol (and displacement, if any).
%ly	ULONG64	The address of a debugger symbol in the target's virtual address space.	A string containing the name of the specified symbol (and displacement, if any), as well as any available source line information.

Arguments

Specifies arguments for the format string, as in **printf**. The number of arguments that are specified should match the number of conversion characters in *FormatString*. Each argument is an expression that will be evaluated by the default expression evaluator (MASM or C++). For details, see Numerical Expression Syntax.

Additional Information

For information about other control flow tokens and their use in debugger command programs, see Using Debugger Command Programs.

Remarks

The color settings that you can choose by using the *Options* parameter are by default all set to black text on a white background. To make best use of these options, you must first use View | Options to open the Options dialog box and change the color settings for Debugger Command window messages.

The following example shows how to include a DML tag in the format string.

```
.printf /D "Click <link cmd=\".chain /D\">here</link> to see extensions DLLs."
```



The output shown in the preceding image has a link that you can click to execute the command specified in the `<link>` tag. The following image shows the result of clicking the link.



For information about DML tags, see dml.doc in the installation folder for Debugging Tools for Windows.

.while

6/25/2019 • 2 minutes to read • Edit Online

The **.while** token behaves like the **while** keyword in C.

```
.while (Condition) { Commands }
```

Syntax Elements

Condition
Specifies a condition. If this evaluates to zero, it is treated as false; otherwise it is true. Enclosing *Condition* in parentheses is optional. *Condition* must be an expression, not a debugger command. It will be evaluated by the default expression evaluator (MASM or C++). For details, see Numerical Expression Syntax.

Commands
Specifies one or more commands that will be executed repeatedly as long as the condition is true. This block of commands needs to be enclosed in braces, even if it consists of a single command. Multiple commands should be separated by semicolons, but the final command before the closing brace does not need to be followed by a semicolon.

Additional Information

For information about other control flow tokens and their use in debugger command programs, see Using Debugger Command Programs.

Remarks

The **.break** and **.continue** tokens can be used to exit or restart the *Commands* block.

Commands

12/5/2018 • 2 minutes to read • Edit Online

This section of the reference discusses the various debugger commands that you can use in CDB, KD, and WinDbg.

ENTER (Repeat Last Command)

12/5/2018 • 2 minutes to read • Edit Online

The ENTER key repeats the last command that you typed.

```
ENTER
```

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

In CDB and KD, pressing the ENTER key by itself at a command prompt reissues the command that you previously entered.

In WinDbg, the ENTER key can have no effect or you can use it to repeat the previous command. You can set this option in the **Options** dialog box. (To open the **Options** dialog box, click **Options** on the **View** menu or click the **Options** button ( ) on the toolbar.)

If you set ENTER to repeat the last command, but you want to create white space in the Debugger Command window, use the *** (Comment Line Specifier)** token and then press ENTER several times.

$<, $><, $$<, $$><, $$ >a< (Run Script File)

12/5/2018 • 4 minutes to read • Edit Online

The **$<**, **$><**, **$$<**, **$$><**, and **$$>a<** commands read the contents of the specified script file and use its contents as debugger command input.

```
$<Filename
$><Filename
$$<Filename
$$><Filename
$$>a<Filename [arg1 arg2 arg3 ...]
```

Parameters

Filename
Specifies a file that contains valid debugger command text. The file name must follow Microsoft Windows file name conventions. The file name may contain spaces.

argn
Specifies any number of string arguments for the debugger to pass to the script. The debugger will replace any string of the form ${$arg*n*} in the script file with the corresponding *argn* before executing the script. Arguments may not contain quotation marks or semicolons. Multiple arguments must be separated by spaces; if an argument contains a space it must be enclosed in quotation marks. All arguments are optional.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

The **$$<** and **$<** tokens execute the commands that are found in the script file literally. However, with **$<** you can specify any file name, including one that contains semicolons. Because **$<** allows semicolons to be used in the file name, you cannot concatenate **$<** with other debugger commands, because a semicolon cannot be used both as a command separator and as part of a file name.

The **$$><** and **$><** tokens execute the commands that are found in the script file literally, which means they open the script file, replace all carriage returns with semicolons, and execute the resulting text as a single command block. As with **$<** discussed previously, the **$><** variation permits file names that contains semicolons, which means you cannot concatenate **$><** with other debugger commands.

The **$$><** and **$><** tokens are useful if you are running scripts that contain debugger command programs. For more information about these programs, see Using Debugger Command Programs.

Unless you have file names that contain semicolons, you do not need to use either **$<** or **$><**.

The **$$>a<** token allows the debugger to pass arguments to the script. If *Filename* contains spaces, it must be enclosed in quotation marks. If too many arguments are supplied, the excess arguments are ignored. If too few arguments are supplied, any token in the source file of the form ${$arg*n*} where *n* is larger than the number of supplied arguments will remain in its literal form and will not be replaced with anything. You can follow this command with a semicolon and additional commands; the presence of a semicolon terminates the argument list.

When the debugger executes a script file, the commands and their output are displayed in the Debugger Command window. When the end of the script file is reached, control returns to the debugger.

The following table summarizes how you can use these tokens.

TOKEN	ALLOWS FILE NAMES THAT CONTAIN SEMICOLONS	ALLOWS CONCATENATION OF ADDITIONAL COMMANDS SEPARATED BY SEMICOLONS	CONDENSES TO SINGLE COMMAND BLOCK	ALLOWS SCRIPT ARGUMENTS
$<	Yes	No	No	No
$><	Yes	No	Yes	No
$$<	No	Yes	No	No
$$><	No	Yes	Yes	No
$$>a<	No	Yes	Yes	Yes

The **$<**, **$><**, **$$<**, and **$$><** commands echo the commands contained in the script file and display the output of these commands. The **$$>a<** command does not echo the commands found in the script file, but merely displays their output.

Script files can be nested. If the debugger encounters one of these tokens in a script file, execution moves to the new script file and returns to the previous location when the new script file has been completed. Scripts can also be called recursively.

In WinDbg, you can paste the additional command text in the Debugger Command window.

Examples

The following example demonstrates how to pass arguments to a script file, Myfile.txt. Assume that the file contains the following text:

```
.echo The first argument is ${$arg1}.
.echo The second argument is ${$arg2}.
```

Then you can pass arguments to this file by using a command like this:

```
0:000> $$>a<myfile.txt myFirstArg mySecondArg
```

The result of this command would be:

```
The first argument is myFirstArg.
The second argument is mySecondArg.
```

Here is an example of what happens when the wrong number of argument is supplied. Assume that the file My Script.txt contains the following text:

```
.echo The first argument is ${$arg1}.
.echo The fifth argument is ${$arg5}.
.echo The fourth argument is ${$arg4}.
```

Then the following semicolon-delimited command line produces output thus:

```
0:000> $$>a< "c:\binl\my script.txt" "First one" Two "Three More" Four; recx
The first argument is First one.
The fifth argument is ${$arg5}.
The fourth argument is Four.
ecx=0021f4ac
```

In the preceding example, the file name is enclosed in quotation marks because it contains a space, and arguments that contain spaces are enclosed in quotation marks as well. Although a fifth argument seems to be expected by the script, the semicolon terminates the **$$>a<** command after the fourth argument.

? (Command Help)

12/5/2018 • 2 minutes to read • Edit Online

The question mark (**?**) character displays a list of all commands and operators.

Note A question mark by itself displays command help. The **? expression** syntax evaluates the given expression.

```
?
```

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

For more information about standard commands, use **?**. For more information about meta-commands, use **.help**.
For more information about extension commands, use **!help**.

? (Evaluate Expression)

12/5/2018 • 2 minutes to read • Edit Online

The question mark (**?**) command evaluates and displays the value of an expression.

Note A question mark by itself (**?**) displays command help. The **?** *expression* command evaluates the given expression.

```
? Expression
```

Parameters

Expression
Specifies the expression to evaluate.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

The input and output of the **?** command depend on whether you are using MASM expression syntax or C++ expression syntax. For more information about these kinds of expression syntax, see Evaluating Expressions and Numerical Expression Syntax.

If you are using MASM syntax, the input and output depend on the current radix. To change the radix, use the **n (Set Number Base)** command.

The **?** command evaluates symbols in the expression in the context of the current thread and process.

Some strings may contain escapes, such as **\n**, **\"**, **\r**, and **\b**, that are meant to be read literally, rather than interpreted by the evaluator. If an escape within a string is interpreted by the evaluator, errors in evaluation can occur. For example:

```
0:000> as AliasName c:\dir\name.txt
0:000> al
  Alias            Value
 -------          -------
 AliasName        c:\dir\name.txt
0:001> ? $spat( "c:\dir\name.txt", "*name*" )
Evaluate expression: 0 = 00000000
0:001> ? $spat( "${AliasName}", "*name*" )
Evaluate expression: 0 = 00000000
0:001> ? $spat( "c:\dir\", "*filename*" )
Syntax error at '( "c:\dir\", "*filename*" )
```

In the first two examples, even though the string does match the pattern, the evaluator is returning a value of **FALSE**. In the third, the evaluator cannot make a comparison because the string ends in a backslash (\ ), and so the **\"** is translated by the evaluator.

To get the evaluator to interpret a string literally, you must use the **@"*String*"** syntax. The following code example shows the correct results:

```
0:000> ? $spat( @"c:\dir\name.txt", "*name*" )
Evaluate expression: 1 = 00000000`00000001
0:000> ? $spat( @"${AliasName}", "*name*" )
Evaluate expression: 1 = 00000000`00000001
0:001> ? $spat( @"c:\dir\", "*filename*" )
Evaluate expression: 0 = 00000000
```

In the preceding examples, the **$spat** MASM operator checks the first string to determine whether it matches (case-insensitive) the pattern of the second string. For more information about MASM operators, see the MASM Numbers and Operators topic.

See also

?? (Evaluate C++ Expression)

.formats (Show Number Formats)

?? (Evaluate C++ Expression)

12/5/2018 • 2 minutes to read • Edit Online

The double question mark (**??**) command evaluates and displays the value of an expression according to C++ expression rules.

```
?? Expression
```

Parameters

Expression
Specifies the C++ expression to evaluate. For more information about the syntax, see C++ Numbers and Operators.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

The **??** command evaluates symbols in the expression in the context of the current thread and process.

If you want to evaluate a part of the **Expression** expression according to MASM expression rules, enclose that part in parentheses and add two at signs (**@@**) before it. For more information about MASM expressions and C++ expressions, see Evaluating Expressions and Numerical Expression Syntax.

See also

? (Evaluate Expression)

.formats (Show Number Formats)

(Search for Disassembly Pattern)

12/5/2018 • 2 minutes to read • Edit Online

The number sign (**#**) command searches for the specified pattern in the disassembly code.

```
# [Pattern] [Address [ L Size ]]
```

Parameters

Pattern
Specifies the pattern to search for in the disassembly code. *Pattern* can contain a variety of wildcard characters and specifiers. For more information about the syntax, see String Wildcard Syntax. If you want to include spaces in *Pattern*, you must enclose the pattern in quotation marks. The pattern is not case sensitive. If you have previously used the **#** command and you omit *Pattern*, the command reuses the most recently used pattern.

Address
Specifies the address where the search begins. For more information about the syntax, see Address and Address Range Syntax.

Size
Specifies the number of instructions to search. If you omit *Size*, the search continues until the first match occurs.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about assembly debugging and related commands, see Debugging in Assembly Mode.

Remarks

If you previously used the **#** command and you omit *Address*, the search begins where the previous search ended.

This command works by searching the disassembled text for the specified pattern. You can use this command to find register names, constants, or any other string that appears in the disassembly output. You can repeat the command without the *Address* parameter to find successive occurrences of the pattern.

You can view disassembly instructions by using the **u (Unassemble)** command or by using the Disassembly window in WinDbg. The disassembly display contains up to four parts: Address offset, Binary code, Assembly language mnemonic, and Assembly language details. The following example shows a possible display.

```
0040116b    45          inc         ebp
0040116c    fc          cld
0040116d    8945b0      mov         eax,[ebp-0x1c]
```

The **#** command can search for text within any single part of the disassembly display. For example, you could use **# eax 0040116b** to find the **mov eax,[ebp-0x1c]** instruction at address 0040116d. The following commands also find this instruction.

```
#  [ebp?0x  0040116b
#  mov  0040116b
#  8945*  0040116b
#  116d  0040116b
```

However, you cannot search for **mov eax*** as a single unit, because **mov** and **eax** appear in different parts of the display. Instead, use **mov*eax**.

As an additional example, you could issue the following command to search for the first reference to the **strlen** function after the entry point **main**.

```
# strlen main
```

Similarly, you could issue the following two commands to find the first **jnz** instruction after address 0x779F9FBA and then find the next **jnz** instruction after that.

```
# jnz 779f9fba#
```

When you omit *Pattern* or *Address*, their values are based on the previous use of the **#** command. If you omit either parameter the first time that you issue the **#** command, no search is performed. However, the values of *Pattern* and *Address* are initialized even in this situation.

If you include *Pattern* or *Address*, its value is set to the entered value. If you omit *Address*, it is initialized to the current value of the program counter. If you omit *Pattern*, it is initialized to an empty pattern.

|| (System Status)

12/5/2018 • 2 minutes to read • Edit Online

The double vertical bar (**||**) command prints status for the specified system or for all systems that you are currently debugging.

Do not confuse this command with the **| (Process Status)** command.

```
|| System
```

Parameters

System
Specifies the system to display. If you omit this parameter, all systems that you are debugging are displayed. For more information about the syntax, see System Syntax.

Environment

Modes	Multiple target debugging
Targets	Live, crash dump
Platforms	All

Remarks

The **||** command is useful only when you are debugging multiple targets. Many, but not all, multiple-target debugging sessions involve multiple systems. For more information about these sessions, see Debugging Multiple Targets.

Each system listing includes the server name and the protocol details. The system that the debugger is running on is identified as **<Local>**.

The following examples show you how to use this command. The following command displays all systems.

```
3:2:005> ||
```

The following command also displays all systems.

```
3:2:005> ||*
```

The following command displays the currently active system.

```
3:2:005> ||.
```

The following command displays the system that had the most recent exception or break.

```
3:2:005> ||#
```

The following command displays system number 2.

```
3:2:005> ||2
```

||s (Set Current System)

12/5/2018 • 2 minutes to read • Edit Online

The **||s** command sets or displays the current system number.

```
||System s
|| s
```

Do not confuse this command with the **s (Search Memory)**, **~s (Change Current Processor)**, **~s (Set Current Thread)**, or **|s (Set Current Process)** command.

Parameters

System
Specifies the system number to activate. For more information about the syntax, see System Syntax.

Environment

Modes	Multiple target debugging
Targets	Live, crash dump
Platforms	All

Remarks

The **||s** command is useful when you are debugging multiple targets or working with multiple dump files. For more information about these kinds of sessions, see Debugging Multiple Targets.

If you use the **||s** syntax, the debugger displays information about the current system.

This command also disassembles the current instruction for the current system, process, and thread.

Note There are complications, when you debug live targets and dump targets together, because commands behave differently for each type of debugging. For example, if you use the **g (Go)** command when the current system is a dump file, the debugger begins executing, but you cannot break back into the debugger, because the break command is not recognized as valid for dump file debugging.

| (Process Status)

12/5/2018 • 2 minutes to read • Edit Online

The pipe (|) command displays status for the specified process, or for all processes that you are currently debugging.

Do not confuse this command with the **|| (System Status)** command.

```
| Process
```

Parameters

Process
Specifies the process to display. If you omit this parameter, all processes that you are debugging are displayed. For more information about the syntax, see Process Syntax.

Environment

Modes	User mode only
Targets	Live, crash dump
Platforms	All

Additional Information

For more information and other methods of displaying or controlling processes and threads, see Controlling Processes and Threads.

Remarks

You can specify processes only in user mode.

You can add a process symbol before many commands. For more information about the meaning of a pipe (|) followed by a command, see the entry for the command itself.

Unless you enabled the debugging of child processes when you started the debugging session, there is only one process that is available to the debugger.

The following examples show you how to use this command. The following command displays all processes.

```
2:005> |
```

The following command also displays all processes.

```
2:005> |*
```

The following command displays the currently active process.

```
2:005> |.
```

The following command displays the process that originally caused the exception (or that the debugger originally attached to).

```
2:005> |#
```

The following command displays process number 2.

```
2:005> |2
```

The previous command displays the following output.

```
0:002> |
#  0 id: 224   name: myprog.exe
   1 id: 228   name: onechild.exe
. 2 id: 22c   name: anotherchild.exe
```

On the first line of this output, 0 is the decimal process number, 224 is the hexadecimal process ID, and *Myprog.exe* is the application name of the process. The period (.) before process 2 means that this process is the current process. The number sign (#) before process 0 means that this process was the one that originally caused the exception or that the debugger attached to.

|s (Set Current Process)

12/5/2018 • 2 minutes to read • Edit Online

The **|s** command sets or displays the current process number.

Do not confuse this command with the **s (Search Memory)**, **~s (Change Current Processor)**, **~s (Set Current Thread)**, or **||s (Set Current System)** command.

```
|Process s
| s
```

Parameters

Process
Specifies the process to set or display. For more information about the syntax, see Process Syntax.

Environment

Modes	User mode only
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about other methods of displaying or controlling processes and threads, see Controlling Processes and Threads.

Remarks

You can specify processes only in user mode.

If you use the **|s** syntax, the debugger displays information about the current process.

This command also disassembles the current instruction for the current system, process, and thread.

~ (Thread Status)

12/5/2018 • 2 minutes to read • Edit Online

The tilde (~) command displays status for the specified thread or for all threads in the current process.

```
~ Thread
```

Parameters

Thread
Specifies the thread to display. If you omit this parameter, all threads are displayed. For more information about the syntax, see Thread Syntax.

Environment

Modes	User mode only
Targets	Live, crash dump
Platforms	All

Additional Information

For more information and other methods of displaying or controlling processes and threads, see Controlling Processes and Threads.

Remarks

You can specify threads only in user mode. In kernel mode, the tilde (~) refers to a processor.

You can add a thread symbol before many commands. For more information about the meaning of a tilde (~) followed by a command, see the entry for the command itself.

The following examples show you how to use this command. The following command displays all threads.

```
0:001> ~
```

The following command also displays all threads.

```
0:001> ~*
```

The following command displays the currently active thread.

```
0:001> ~.
```

The following command displays the thread that originally caused the exception (or that was active when the debugger attached to the process).

```
0:001> ~#
```

The following command displays thread number 2.

```
0:001> ~2
```

The previous command displays the following output.

```
0:001> ~
   0 id: 4dc.470 Suspend: 0 Teb 7ffde000 Unfrozen
. 1 id: 4dc.534 Suspend: 0 Teb 7ffdd000 Unfrozen
#  2 id: 4dc.5a8 Suspend: 0 Teb 7ffdc000 Unfrozen
```

On the first line of this output, 0 is the decimal thread number, 4DC is the hexadecimal process ID, 470 is the hexadecimal thread ID, 0x7FFDE000 is the address of the TEB, and **Unfrozen** is the thread status. The period (.) before thread 1 means this thread is the current thread. The number sign (#) before thread 2 means this thread was the one that originally caused the exception or it was active when the debugger attached to the process.

~e (Thread-Specific Command)

12/5/2018 • 2 minutes to read • Edit Online

The **~e** command executes one or more commands for a specific thread or for all threads in the target process.

Do not confuse this command with the **e (Enter Values)** command.

```
~Thread e CommandString
```

Parameters

Thread
Specifies the thread or threads that the debugger will execute *CommandString* for. For more information about the syntax, see Thread Syntax.

CommandString
Specifies one or more commands to execute. You should separate multiple commands by using semicolons. *CommandString* includes the rest of the input line. All of the text that follows the letter "e" is interpreted as part of this string. Do not enclose *CommandString* in quotation marks.

Environment

Modes	User mode only
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about other commands that control threads, see Controlling Processes and Threads.

Remarks

You can specify threads only in user mode. In kernel mode, the tilde (~) refers to a processor.

When you use the **~e** command together with one thread, the **~e** command only saves some typing. For example, the following two commands are equivalent.

```
0:000> ~2e r; k; kd

0:000> ~2r; ~2k; ~2kd
```

However, you can use the **~e** qualifier to repeat a command or extension command several times. When you use the qualifier in this manner, it can eliminate extra typing. For example, the following command repeats the **!gle** extension command for every thread that you are debugging.

```
0:000> ~*e !gle
```

If an error occurs in the execution of one command, execution continues with the next command.

You cannot use the **~e** qualifier together with execution commands (**g**, **gh**, **gn**, **gN**, **gu**, **p**, **pa**, **pc**, **t**, **ta**, **tb**, **tc**, **wt**).

You cannot use the **~e** qualifier together with the **j (Execute If-Else)** or **z (Execute While)** conditional commands.

If you are debugging more than one process, you cannot use the **~e** command to access the virtual memory space for a inactive process.

~f (Freeze Thread)

12/5/2018 • 2 minutes to read • Edit Online

The **~f** command freezes the given thread, causing it to stop and wait until it is unfrozen.

Do not confuse this command with the **f (Fill Memory)** command.

```
~Thread f
```

Parameters

Thread
Specifies the thread to freeze. For more information about the syntax, see Thread Syntax.

Environment

Modes	User mode only
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about how frozen threads behave and a list of other commands that control the freezing and suspending of threads, see Controlling Processes and Threads.

Remarks

You can specify threads only in user mode. In kernel mode, the tilde (~) refers to a processor.

The **~f** command causes the specified thread to freeze. When the debugger enables the target application to resume execution, other threads execute as expected while this thread remains stopped.

The following examples show you how to use this command. The following command displays the current status of all threads.

```
0:000> ~* k
```

The following command freezes the thread that caused the current exception.

```
0:000> ~# f
```

The following command checks that the status of this thread is suspended.

```
0:000> ~* k
```

The following command unfreezes thread number 123.

```
0:000> ~123 u
```

~u (Unfreeze Thread)

12/5/2018 • 2 minutes to read • Edit Online

The **~u** command unfreezes the specified thread.

Do not confuse this command with the **U (Unassemble)** command.

```
~Thread u
```

Parameters

Thread
Specifies the thread or threads to unfreeze. For more information about the syntax, see Thread Syntax.

Environment

Modes	User mode only
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about how frozen threads behave and a list of other commands that control the freezing and suspending of threads, see Controlling Processes and Threads.

Remarks

You can specify threads only in user mode. In kernel mode, the tilde (~) refers to a processor.

The following examples show you how to use the ~ commands.

The following command displays the current status of all threads.

```
0:000> ~* k
```

The following command freeze the thread that caused the current exception.

```
0:000> ~# f
```

The following command checks that the status of this thread is suspended.

```
0:000> ~* k
```

The following command unfreezes thread number 123.

```
0:000> ~123 u
```

~n (Suspend Thread)

12/5/2018 • 2 minutes to read • Edit Online

The **~n** command suspends execution of the specified thread.

Do not confuse this command with the **n (Set Number Base)** command.

```
~Thread n
```

Parameters

Thread
Specifies the thread or threads to suspend. For more information about the syntax, see Thread Syntax.

Environment

Modes	User mode only
Targets	Live debugging only
Platforms	All

Additional Information

For more information about the suspend count and how suspended threads behave and for a list of other commands that control the suspending and freezing of threads, see Controlling Processes and Threads.

Remarks

You can specify threads only in user mode. In kernel mode, the tilde (~) refers to a processor.

Every time that you use the **~n** command, the thread's suspend count is increased by one.

The thread's start address is displayed when you use this command.

~m (Resume Thread)

12/5/2018 • 2 minutes to read • Edit Online

The **~m** command resumes execution of the specified thread.

Do not confuse this command with the **m (Move Memory)** command.

```
~Thread m
```

Parameters

Thread
Specifies the thread or threads to resume. For more information about the syntax, see Thread Syntax.

Environment

Modes	User mode only
Targets	Live debugging only
Platforms	All

Additional Information

For more information about the suspend count and how suspended threads behave and for a list of other commands that control the suspending and freezing of threads, see Controlling Processes and Threads.

Remarks

You can specify threads only in user mode. In kernel mode, the tilde (~) refers to a processor.

Every time that you use the **~m** command, the thread's suspend count is decreased by one.

~s (Set Current Thread)

12/5/2018 • 2 minutes to read • Edit Online

The **~s** command sets or displays the current thread number.

In user mode, **~s** sets the current thread. Do not confuse this command confused with the **~s (Change Current Processor)** command (which works only in kernel mode), the **|s (Set Current Process)** command, the **||s (Set Current System)** command, or the **s (Search Memory)** command.

```
~Thread s
~ s
```

Parameters

Thread
Specifies the thread to set or display. For more information about the syntax, see Thread Syntax.

Environment

Modes	User mode only
Targets	Live, crash dump
Platforms	All

Additional Information

For more information and other methods of displaying or controlling processes and threads, see Controlling Processes and Threads.

Remarks

You can specify threads only in user mode. In kernel mode, the tilde (~) refers to a processor.

If you use the **~s** syntax, the debugger displays information about the current thread.

This command also disassembles the current instruction for the current system, process, and thread.

~s (Change Current Processor)

12/5/2018 • 2 minutes to read • Edit Online

The **~s** command sets which processor is debugged on a multiprocessor system.

In kernel mode, **~s** changes the current processor. Do not confuse this command with the **~s (Set Current Thread)** command (which works only in user mode), the **|s (Set Current Process)** command, the **||s (Set Current System)** command, or the **s (Search Memory)** command.

```
~Processor s
```

Parameters

Processor
Specifies the number of the processor to debug.

Environment

Modes	Kernel mode only
Targets	Live, crash dump
Platforms	All

Remarks

You can specify processors only in kernel mode. In user mode, the tilde (~) refers to a thread.

You can immediately tell when you are working on a multiple processor system by the shape of the kernel debugging prompt. In the following example, 0: means that you are debugging the first processor in the computer.

```
0: kd>
```

Use the following command to switch between processors:

```
0: kd> ~1s
1: kd>
```

Now the second processor in the computer that is being debugged.

See also

Multiprocessor Syntax

a (Assemble)

12/5/2018 • 2 minutes to read • Edit Online

The **a** command assembles 32-bit x86 instruction mnemonics and puts the resulting instruction codes into memory.

```
a [Address]
```

Parameters

Address
Specifies the beginning of the block in memory where the resulting codes are put. For more information about the syntax, see Address and Address Range Syntax.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about assembly debugging and related commands, see Debugging in Assembly Mode.

Remarks

The **a** command does not support 64-bit instruction mnemonics. However, the **a** command is enabled regardless of whether you are debugging a 32-bit target or a 64-bit target. Because of the similarities between x86 and x64 instructions, you can sometimes use the **a** command successfully when debugging a 64-bit target.

If you do not specify an address, the assembly starts at the address that the current value of the instruction pointer specifies. To assemble a new instruction, type the desired mnemonic and press ENTER. To end assembly, press only ENTER.

Because the assembler searches for all of the symbols that are referred to in the code, this command might take time to complete. During this time, you cannot press **CTRL+C** to end the **a** command.

ad (Delete Alias)

12/5/2018 • 2 minutes to read • Edit Online

The **ad** command deletes an alias from the alias list.

```
ad [/q] Name
ad *
```

Parameters

/q
Specifies quiet mode. This mode hides the error message if the alias that *Name* specifies does not exist.

Name
Specifies the name of the alias to delete. If you specify an asterisk (*), all aliases are deleted (even if there is an alias whose name is "*").

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about how to use aliases, see Using Aliases.

Remarks

You can use the **ad** command to delete any user-named alias. But you cannot use this command to delete a fixed-name alias ($u0 to $u9).

ah (Assertion Handling)

12/5/2018 • 2 minutes to read • Edit Online

The **ah** command controls the assertion handling status for specific addresses.

```
ahb [Address]
ahi [Address]
ahd [Address]
ahc
ah
```

Parameters

ahb
Breaks into the debugger if an assertion fails at the specified address.

ahi
Ignores an assertion failure at the specified address.

ahd
Deletes any assertion-handling information at the specified address. This deletion causes the debugger to return to its default state for that address.

Address
Specifies the address of the instruction whose assertion-handling status is being set. If you omit this parameter, the debugger uses the current program counter.

ahc
Deletes all assertion-handling information for the current process.

ah
Displays the current assertion-handling settings.

Environment

Modes	User mode, kernel mode
Targets	Live debugging only
Platforms	All

Additional Information

For more information about break status and handling status, descriptions of all event codes, a list of the default status for all events, and details about other methods of controlling this status, see Controlling Exceptions and Events.

Remarks

The **ah*** command controls the assertion handling status for a specific address. The **sx* asrt** command controls the global assertion handling status. If you use **ah*** for a certain address and then an assert occurs there, the

debugger responds based on the **ah*** settings and ignores the **sx* asrt** settings.

When the debugger encounters an assertion, the debugger first checks whether handling has been configured for that specific address. If you have not configured the handling, the debugger uses the global setting.

The **ah*** command affects only the current process. When the current process ends, all status settings are lost.

Assertion handling status affects only STATUS_ASSERTION_EXCEPTION exceptions. This handling does not affect the kernel-mode ASSERT routine.

al (List Aliases)

12/5/2018 • 2 minutes to read • Edit Online

The **al** command displays a list of all currently defined user-named aliases.

```
al
```

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about how to use aliases, see Using Aliases.

Remarks

The **al** command lists all user-named aliases. But this command does not list fixed-name aliases ($u0 to $u9).

as, aS (Set Alias)

12/5/2018 • 4 minutes to read • Edit Online

The **as** and **aS** commands define a new alias or redefine an existing one.

```
as Name EquivalentLine
aS Name EquivalentPhrase
aS Name "EquivalentPhrase"
as /e Name EnvironmentVariable
as /ma Name Address
as /mu Name Address
as /msa Name Address
as /msu Name Address
as /x Name Expression
aS /f Name File
as /c Name CommandString
```

Parameters

Name
Specifies the alias name. This name can be any text string that does not contain a space or the ENTER keystroke and does not begin with "al", "as", "aS", or "ad". *Name* is case sensitive.

EquivalentLine
Specifies the alias equivalent. *EquivalentLine* is case sensitive. You must add at least one space between *Name* and *EquivalentLine*. The number of spaces between these two parameters is not important. The alias equivalent never contains leading spaces. After these spaces, *EquivalentLine* includes the rest of the line. Semicolons, quotation marks, and spaces are treated as literal characters, and trailing spaces are included.

EquivalentPhrase
Specifies the alias equivalent. *EquivalentPhrase* is case sensitive. You must add at least one space between *Name* and *EquivalentPhrase*. The number of spaces between these two parameters is not important. The alias equivalent never contains leading spaces.

You can enclose *EquivalentPhrase* in quotation marks ("). Regardless of whether you use quotation marks, *EquivalentPhrase* can contain spaces, commas, and single quotation marks ('). If you enclose *EquivalentPhrase* in quotation marks, it can include semicolons, but not additional quotation marks. If you do not enclose *EquivalentPhrase* in quotation marks, it can include quotation marks in any location other than the first character, but it cannot include semicolons. Trailing spaces are included regardless of whether you use quotation marks.

/e
Sets the alias equivalent equal to the environment variable that *EnvironmentVariable* specifies.

EnvironmentVariable
Specifies the environment variable that is used to determine the alias equivalent. The debugger's environment is used, not the target's. If you started the debugger at a Command Prompt window, the environment variables in that window are used.

/ma
Sets the alias equivalent equal to the null-terminated ASCII string that begins at *Address*.

/mu
Sets the alias equivalent equal to the null-terminated Unicode string that begins at *Address*.

/msa
Sets the alias equivalent equal to the ANSI_STRING structure that is located at *Address*.

/msu
Sets the alias equivalent equal to the UNICODE_STRING structure that is located at *Address*.

Address
Specifies the location of the virtual memory that is used to determine the alias equivalent.

/x
Sets the alias equivalent equal to the 64-bit value of *Expression*.

Expression
Specifies the expression to evaluate. This value becomes the alias equivalent. For more information about the syntax, see Numerical Expression Syntax.

/f
Sets the alias equivalent equal to the contents of the *File* file. You should always use the **/f** switch together with **aS**, not with **as**.

File
Specifies the file whose contents become the alias equivalent. *File* can contain spaces, but you should never enclose *File* in quotation marks. If you specify an invalid file, you receive an "Out of memory" error message.

/c
Sets the alias equivalent equal to the output of the commands that *CommandString* specifies. The alias equivalent includes carriage returns if they are present within the command display and a carriage return at the end of the display of each command (even if you specify only one command).

CommandString
Specifies the commands whose outputs become the alias equivalent. This string can include any number of commands that are separated by semicolons.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about how to use aliases, see Using Aliases.

Remarks

If you do not use any switches, the **as** command uses the rest of the line as the alias equivalent.

You can end the **aS** command by a semicolon. This technique is useful in a script when you have to put all commands on a single line. Note that if the portion of the line after the semicolon requires expansion of the alias, you must enclose that second portion of the line in a new block. The following example produces the expected output, 0x6.

```
0:001> aS /x myAlias 5 + 1; .block{.echo myAlias}
0x6
```

If you omit the new block, you do not get the expected output. That is because the expansion of a newly set alias does not happen until a new code block is entered. In the following example, the new block is omitted, and the output is the text "myAlias" instead of the expected value 0x6.

```
0:001> aS /x myAlias 5 + 1; .echo myAlias
myAlias
```

For more information about using aliases in scripts, see Using Aliases.

If you use a **/e**, **/ma**, **/mu**, **/msa**, **/msu**, or **/x** switch, the **as** and **aS** commands work the same and the command ends if a semicolon is encountered.

If *Name* is already the name of an existing alias, that alias is redefined.

You can use the **as** or **aS** command to create or change any user-named alias. But you cannot use the command to control a fixed-name alias ($u0 to $u9).

You can use the **/ma**, **/mu**, **/msa**, **/msu**, **/f**, and **/c** switches to create an alias that contains carriage returns. However, you cannot use an alias that contains carriage returns to execute multiple commands in sequence. Instead, you must use semicolons.

ba (Break on Access)

1/3/2019 • 6 minutes to read • Edit Online

The **ba** command sets a processor breakpoint (often called, less accurately, a *data breakpoint*). This breakpoint is triggered when the specified memory is accessed.

User-Mode

```
[~Thread] ba[ID] Access Size [Options] [Address [Passes]] ["CommandString"]
```

Kernel-Mode

```
ba[ID] Access Size [Options] [Address [Passes]] ["CommandString"]
```

Parameters

Thread
Specifies the thread that the breakpoint applies to. For more information about syntax, see Thread Syntax. You can specify threads only in user mode.

ID
Specifies an optional number that identifies the breakpoint. If you do not specify *ID*, the first available breakpoint number is used. You cannot add space between **ba** and the ID number. Each processor supports only a limited number of processor breakpoints, but there is no restriction on the value of the *ID* number. If you enclose *ID* in square brackets ([]), *ID* can include any expression. For more information about the syntax, see Numerical Expression Syntax.

Access
Specifies the type of access that satisfies the breakpoint. This parameter can be one of the following values.

OPTION	ACTION
e (execute)	Breaks into the debugger when the CPU retrieves an instruction from the specified address.
r (read/write)	Breaks into the debugger when the CPU reads or writes at the specified address.
w (write)	Breaks into the debugger when the CPU writes at the specified address.
i (i/o)	(Kernel mode only, x86-based systems only) Breaks into the debugger when the I/O port at the specified *Address* is accessed.

Size
Specifies the size of the location, in bytes, to monitor for access. On an x86-based processor, this parameter can

be 1, 2, or 4. However, if *Access* equals **e**, *Size* must be 1.

On an x64-based processor, this parameter can be 1, 2, 4, or 8. However, if *Access* equals **e**, *Size* must be 1.

Options
Specifies breakpoint options. You can use any number of the following options, except as indicated:

/1
Creates a "one-shot" breakpoint. After this breakpoint is triggered, the breakpoint is permanently removed from the breakpoint list.

/p *EProcess*
(Kernel mode only) Specifies a process that is associated with this breakpoint. *EProcess* should be the actual address of the EPROCESS structure, not the PID. The breakpoint is triggered only if it is encountered in the context of this process.

/t *EThread*
(Kernel mode only) Specifies a thread that is associated with this breakpoint. *EThread* should be the actual address of the ETHREAD structure, not the thread ID. The breakpoint is triggered only if it is encountered in the context of this thread. If you use **/p** *EProcess* and **/t** *EThread* , you can enter them in either order.

/c *MaxCallStackDepth*
Causes the breakpoint to be active only when the call stack depth is less than *MaxCallStackDepth*. You cannot combine this option together with **/C**.

/C *MinCallStackDepth*
Causes the breakpoint to be active only when the call stack depth is larger than *MinCallStackDepth*. You cannot combine this option together with **/c**.

Address
Specifies any valid address. If the application accesses memory at this address, the debugger stops execution and displays the current values of all registers and flags. This address must be an offset and suitably aligned to match the *Size* parameter. (For example, if *Size* is 4, *Address* must be a multiple of 4.) If you omit *Address*, the current instruction pointer is used. For more information about the syntax, see Address and Address Range Syntax.

Passes
Specifies the number of times the breakpoint is passed by until it activates. This number can be any 16-bit value. The number of times the program counter passes through this point without breaking is one less than the value of this number. Therefore, omitting this number is the same as setting it equal to 1. Note also that this number counts only the times that the application *executes* past this point. Stepping or tracing past this point does not count. After the full count is reached, you can reset this number only by clearing and resetting the breakpoint.

CommandString
Specifies a list of commands to execute every time that the breakpoint is encountered the specified number of times. These commands are executed only if the breakpoint is hit after you issue a **g (Go)** command, instead of after a **t (Trace)** or **p (Step)** command. Debugger commands in *CommandString* can include parameters.

You must enclose this command string in quotation marks, and you should separate multiple commands by semicolons. You can use standard C control characters (such as **\n** and **\"**). Semicolons that are contained in second-level quotation marks (**\"**) are interpreted as part of the embedded quoted string.

This parameter is optional.

Environment

Modes	User mode, kernel mode

Targets	Live debugging only
Platforms	All

Additional Information

For more information on processor breakpoints, see Processor Breakpoints (ba Breakpoints). For more information about and examples of using breakpoints, other breakpoint commands and methods of controlling breakpoints, and information about how to set breakpoints in user space from a kernel debugger, see Using Breakpoints. For more information about conditional breakpoints, see Setting a Conditional Breakpoint.

Remarks

The debugger uses the *ID* number to refer to the breakpoint in later **bc (Breakpoint Clear)**, **bd (Breakpoint Disable)**, and **be (Breakpoint Enable)** commands.

Use the **bl (Breakpoint List)** command to list all existing breakpoints, their ID numbers, and their status.

Use the **.bpcmds (Display Breakpoint Commands)** command to list all existing breakpoints, their ID numbers, and the commands that were used to create them.

Each processor breakpoint has a size associated with it. For example, a **w** (write) processor breakpoint could be set at the address 0x70001008 with a size of four bytes. This would monitor the block of addresses from 0x70001008 to 0x7000100B, inclusive. If this block of memory is written to, the breakpoint will be triggered.

It can happen that the processor performs an operation on a memory region that *overlaps* with, but is not identical to, the specified region. In this example, a single write operation that includes the range 0x70001000 to 0x7000100F, or a write operation that includes only the byte at 0x70001009, would be an overlapping operation. In such a situation, whether the breakpoint is triggered is processor-dependent. You should consult the processor manual for specific details. To take one specific instance, on an x86 processor, a read or write breakpoint is triggered whenever the accessed range overlaps the breakpoint range.

Similarly, if an **e** (execute) breakpoint is set on the address 0x00401003, and then a two-byte instruction spanning the addresses 0x00401002 and 0x00401003 is executed, the result is processor-dependent. Again, consult the processor architecture manual for details.

The processor distinguishes between breakpoints set by a user-mode debugger and breakpoints set by a kernel-mode debugger. A user-mode processor breakpoint does not affect any kernel-mode processes. A kernel-mode processor breakpoint might or might not affect a user-mode process, depending on whether the user-mode code is using the debug register state and whether there is a user-mode debugger that is attached.

To apply the current process' existing data breakpoints to a different register context, use the **.apply_dbp (Apply Data Breakpoint to Context)** command.

On a multiprocessor computer, each processor breakpoint applies to all processors. For example, if the current processor is 3 and you use the command `ba e1 MyAddress` to put a breakpoint at MyAddress, any processor -- not only processor 3 -- that executes at that address triggers the breakpoint. (This holds for software breakpoints as well.)

You cannot create multiple processor breakpoints at the same address that differ only in their *CommandString* values. However, you can create multiple breakpoints at the same address that have different restrictions (for example, different values of the **/p**, **/t**, **/c**, and **/C** options).

For more details on processor breakpoints, and additional restrictions that apply to them, see Processor Breakpoints (ba Breakpoints).

The following examples show the **ba** command. The following command sets a breakpoint for read access on 4 bytes of the variable myVar.

```
0:000> ba r4 myVar
```

The following command adds a breakpoint on all serial ports with addresses from 0x3F8 through 0x3FB. This breakpoint is triggered if anything is read or written to these ports.

```
kd> ba i4 3f8
```

bc (Breakpoint Clear)

12/5/2018 • 2 minutes to read • Edit Online

The **bc** command permanently removes previously set breakpoints from the system.

```
bc Breakpoints
```

Parameters

Breakpoints
Specifies the ID numbers of the breakpoints to remove. You can specify any number of breakpoints. You must separate multiple IDs by spaces or commas. You can specify a range of breakpoint IDs by using a hyphen (-). You can use an asterisk (*) to indicate all breakpoints. If you want to use a numeric expression for an ID, enclose it in brackets ([]). If you want to use a string with wildcard characters to match a breakpoint's symbolic name, enclose it in quotation marks (" ").

Environment

Modes	User mode, kernel mode
Targets	Live debugging only
Platforms	All

Additional Information

For more information about how to use breakpoints, other breakpoint commands and methods of controlling breakpoints, and how to set breakpoints in user space from a kernel debugger, see Using Breakpoints. For more information about conditional breakpoints, see Setting a Conditional Breakpoint.

Remarks

Use the **bl (Breakpoint List)** command to list all existing breakpoints, their ID numbers, and their status.

Use the **.bpcmds (Display Breakpoint Commands)** command to list all existing breakpoints, their ID numbers, and the commands that were used to create them.

bd (Breakpoint Disable)

12/5/2018 • 2 minutes to read • Edit Online

The **bd** command disables, but does not delete, previously set breakpoints.

```
bd Breakpoints
```

Parameters

Breakpoints
Specifies the ID numbers of the breakpoints to disable. You can specify any number of breakpoints. You must separate multiple IDs by spaces or commas. You can specify a range of breakpoint IDs by using a hyphen (-). You can use an asterisk (*) to indicate all breakpoints. If you want to use a numeric expression for an ID, enclose it in brackets ([]). If you want to use a string with wildcard characters to match a breakpoint's symbolic name, enclose it in quotation marks (" ").

Environment

Modes	User mode, kernel mode
Targets	Live debugging only
Platforms	All

Additional Information

For more information about how to use breakpoints, other breakpoint commands and methods of controlling breakpoints, and how to set breakpoints in user space from a kernel debugger, see Using Breakpoints. For more information about conditional breakpoints, see Setting a Conditional Breakpoint.

Remarks

When a breakpoint is disabled, the system does not check whether the conditions that are specified in the breakpoint are valid.

Use the **be (Breakpoint Enable)** command to re-enable a disabled breakpoint.

Use the **bl (Breakpoint List)** command to list all existing breakpoints, their ID numbers, and their status.

Use the **.bpcmds (Display Breakpoint Commands)** command to list all existing breakpoints, their ID numbers, and the commands that were used to create them.

be (Breakpoint Enable)

9/19/2019 • 2 minutes to read • Edit Online

The **be** command restores one or more breakpoints that were previously disabled.

```
be Breakpoints
```

Parameters

Breakpoints
Specifies the ID numbers of the breakpoints to enable. You can specify any number of breakpoints. You must separate multiple IDs by spaces or commas. You can specify a range of breakpoint IDs by using a hyphen (-). You can use an asterisk (*) to indicate all breakpoints. If you want to use a numeric expression for an ID, enclose it in brackets ([ ]). If you want to use a string with wildcard characters to match a breakpoint's symbolic name, enclose it in quotation marks (" ").

Environment

Modes	User mode, kernel mode
Targets	Live debugging only
Platforms	All

Additional Information

For more information about and examples of how to use breakpoints, other breakpoint commands and methods of controlling breakpoints, and how to set breakpoints in user space from a kernel debugger, see Using Breakpoints. For more information about conditional breakpoints, see Setting a Conditional Breakpoint.

Remarks

Use the **bl (Breakpoint List)** command to list all existing breakpoints, their ID numbers, and their status.

Use the **.bpcmds (Display Breakpoint Commands)** command to list all existing breakpoints, their ID numbers, and the commands that were used to create them.

bl (Breakpoint List)

9/25/2019 • 3 minutes to read • Edit Online

The **bl** command lists information about existing breakpoints.

```
bl [/L] [Breakpoints]
```

Parameters

/L
Forces **bl** to always display breakpoint addresses instead of showing source file and line numbers.

Breakpoints
Specifies the ID numbers of the breakpoints to list. If you omit *Breakpoints*, the debugger lists all breakpoints. You can specify any number of breakpoints. You must separate multiple IDs by spaces or commas. You can specify a range of breakpoint IDs by using a hyphen (-). You can use an asterisk (*) to indicate all breakpoints. If you want to use a numeric expression for an ID, enclose it in brackets ([]). If you want to use a string with wildcard characters to match a breakpoint's symbolic name, enclose it in quotation marks ("").

Environment

Modes	User mode, kernel mode
Targets	Live debugging only
Platforms	All

Additional Information

For more information about and examples of how to use breakpoints, other breakpoint commands and methods of controlling breakpoints, and how to set breakpoints in user space from a kernel debugger, see Using Breakpoints. For more information about conditional breakpoints, see Setting a Conditional Breakpoint.

Remarks

For each breakpoint, the command displays the following information:

- The breakpoint ID. This ID is a decimal number that you can use to refer to the breakpoint in later commands.
- The breakpoint status. The status can be **e** (enabled) or **d** (disabled).
- (Unresolved breakpoints only) The letter "u" appears if the breakpoint is unresolved. That is, the breakpoint does not match a symbolic reference in any currently loaded module. For information about these breakpoints, see Unresolved Breakpoints (bu Breakpoints).
- The virtual address or symbolic expression that makes up the breakpoint location. If you enabled source line number loading, the **bl** command displays file and line number information instead of address offsets. If the breakpoint is unresolved, the address is omitted here and appears at the end of the listing instead.

- (Data breakpoints only) Type and size information are displayed for data breakpoints. The types can be **e** (execute), **r** (read/write), **w** (write), or **i** (input/output). These types are followed with the size of the block, in bytes. For information about these breakpoints, see Processor Breakpoints (ba Breakpoints).
- The number of passes that remain until the breakpoint is activated, followed by the initial number of passes in parentheses. For more information about this kind of breakpoint, see the description of the *Passes* parameter in **bp, bu, bm (Set Breakpoint)**.
- The associated process and thread. If thread is given as three asterisks (***), this breakpoint is not a thread-specific breakpoint.
- The module and function, with offset, that correspond to the breakpoint address. If the breakpoint is unresolved, the breakpoint address appears here instead, in parentheses. If the breakpoint is set on a valid address but symbol information is missing, this field is blank.
- The command that is automatically executed when this breakpoint is hit. This command is displayed in quotation marks.

If you are not sure what command was used to set an existing breakpoint, use **.bpcmds (Display Breakpoint Commands)** to list all breakpoints along with the commands that were used to create them.

The following example shows the output of a **bl** command.

Example

```
0:000> bl
 0 e 010049e0     0001 (0001)  0:**** stst!main
```

This output contains the following information:

- The breakpoint ID is **0**.
- The breakpoint status is **e** (enabled).
- The breakpoint is not unresolved (there is no **u** in the output).
- The virtual address of the breakpoint is **010049e0**.
- The breakpoint is active on the first pass through the code and the code has not yet been executed under the debugger. This information is indicated by a value of 1 (**0001**) in the "passes remaining" counter and a value of 1 (**(0001)**) in the initial passes counter.
- This breakpoint is not a thread-specific breakpoint (***).
- The breakpoint is set on **main** in the **stst** module.

bp, bu, bm (Set Breakpoint)

5/30/2019 • 11 minutes to read • Edit Online

The **bp**, **bu**, and **bm** commands set one or more software breakpoints. You can combine locations, conditions, and options to set different kinds of software breakpoints.

User-Mode

```
[~Thread] bp[ID] [Options] [Address [Passes]] ["CommandString"]
[~Thread] bu[ID] [Options] [Address [Passes]] ["CommandString"]
[~Thread] bm [Options] SymbolPattern [Passes] ["CommandString"]
```

Kernel-Mode

```
bp[ID] [Options] [Address [Passes]] ["CommandString"]
bu[ID] [Options] [Address [Passes]] ["CommandString"]
bm [Options] SymbolPattern [Passes] ["CommandString"]
```

Parameters

Thread
Specifies the thread that the breakpoint applies to. For more information about the syntax, see Thread Syntax. You can specify threads only in user mode. If you do not specify a thread, the breakpoint applies to all threads.

ID
Specifies a decimal number that identifies a breakpoint.

The debugger assigns the *ID* when it creates the breakpoint, but you can change it by using the **br (Breakpoint Renumber)** command. You can use the *ID* to refer to the breakpoint in later debugger commands. To display the *ID* of a breakpoint, use the **bl (Breakpoint List)** command.

When you use *ID* in a command, do not type a space between the command (**bp** or **bu**) and the ID number.

The *ID* parameter is always optional. If you do not specify *ID*, the debugger uses the first available breakpoint number. In kernel mode, you can set only 32 breakpoints. In user mode, you can set any number of breakpoints. In either case, there is no restriction on the value of the *ID* number. If you enclose *ID* in square brackets (**[]**), *ID* can include any expression. For more information about the syntax, see Numerical Expression Syntax.

Options
Specifies breakpoint options. You can specify any number of the following options, except as indicated:

/1
Creates a "one-shot" breakpoint. After this breakpoint is triggered, it is deleted from the breakpoint list.

/f *PredNum*
(Itanium-based only, user mode only) Specifies a predicate number. The breakpoint is predicated with the corresponding predicate register. (For example, **bp /f 4** *address* sets a breakpoint that is predicated with the **p4** predicate register.)

/p *EProcess*

(Kernel-mode only) Specifies a process that is associated with this breakpoint. *EProcess* should be the actual address of the EPROCESS structure, not the PID. The breakpoint is triggered only if it is encountered in the context of this process.

/t *EThread*
(Kernel-mode only) Specifies a thread that is associated with this breakpoint. *EThread* should be the actual address of the ETHREAD structure, not the thread ID. The breakpoint is triggered only if it is encountered in the context of this thread. If you use **/p** *EProcess* and **/t** *EThread*, you can enter them in any order.

/c *MaxCallStackDepth*
Activates the breakpoint only when the call stack depth is less than *MaxCallStackDepth*. You cannot use this option together with **/C**.

/C *MinCallStackDepth*
Activates the breakpoint only when the call stack depth is larger than *MinCallStackDepth*. You cannot use this option together with **/c**.

/a
(For **bm** only) Sets breakpoints on all of the specified locations, whether they are in data space or code space. Because breakpoints on data can cause program failures, use this option only on locations that are known to be safe.

/d
(For **bm** only) Converts the breakpoint locations to addresses. Therefore, if the code is moved, the breakpoints remain at the same address, instead of being set according to *SymbolPattern*. Use **/d** to avoid reevaluating changes to breakpoints when modules are loaded or unloaded.

/(
(For **bm** only) Includes parameter list information in the symbol string that *SymbolString* defines.

This feature enables you to set breakpoints on overloaded functions that have the same name but different parameter lists. For example, bm /(myFunc sets breakpoints on both **myFunc(int a)** and **myFunc(char a)**. Without "/(", a breakpoint that is set on **myFunc** fails because it does not indicate which **myFunc** function the breakpoint is intended for.

/w dx object expression Sets a conditional breakpoint based on the boolean value returned by dx object expression. The argument is a data model (dx) expression which evaluates to true (matches condition – break) or false (does not match condition – do not break).

This example sets a conditional breakpoint based on the value of localVariable.

```
bp /w "localVariable == 4" mymodule!myfunction
```

This example shows how to set a breakpoint using JavaScript.

```
bp /w "@$scriptContents.myFunc(localVariable)" @rip
```

For more information on debugger objects, see dx (Display Debugger Object Model Expression).

> **NOTE**
> The /w option is experimental and requires the latest version of the debugger.

Address
Specifies the first byte of the instruction where the breakpoint is set. If you omit *Address*, the current

instruction pointer is used. For more information about the syntax, see Address and Address Range Syntax.

Passes
Specifies the number of the execution pass that the breakpoint is activated on. The debugger skips the breakpoint location until it reaches the specified pass. The value of *Passes* can be any 16-bit or 32-bit value.

By default, the breakpoint is active the first time that the application executes the code that contains the breakpoint location. This default situation is equivalent to a value of **1** for *Passes*. To activate the breakpoint only after the application executes the code at least one time, enter a value of **2** or more. For example, a value of **2** activates the breakpoint the second time that the code is executed.

This parameter creates a counter that is decremented on each pass through the code. To see the initial and current values of the *Passes* counter, use **bl (Breakpoint List)**.

The *Passes* counter is decremented only when the application *executes* past the breakpoint in response to a **g (Go)** command. The counter is not decremented if you are stepping through the code or tracing past it. When the *Passes* counter reaches **1**, you can reset it only by clearing and resetting the breakpoint.

CommandString
Specifies a list of commands that are executed every time that the breakpoint is encountered the specified number of times. You must enclose the *CommandString* parameter in quotation marks. Use semicolons to separate multiple commands.

Debugger commands in *CommandString* can include parameters. You can use standard C-control characters (such as **\n** and **\"**). Semicolons that are contained in second-level quotation marks (**\"**) are interpreted as part of the embedded quoted string.

The *CommandString* commands are executed only if the breakpoint is reached while the application is *executing* in response to a **g (Go)** command. The commands are not executed if you are stepping through the code or tracing past this point.

Any command that resumes program execution after a breakpoint (such as **g** or **t**) ends the execution of the command list.

SymbolPattern
Specifies a pattern. The debugger tries to match this pattern to existing symbols and to set breakpoints on all pattern matches. *SymbolPattern* can contain a variety of wildcard characters and specifiers. For more information about this syntax, see String Wildcard Syntax. Because these characters are being matched to symbols, the match is not case sensitive, and a single leading underscore (_) represents any quantity of leading underscores.

Environment

Modes	User mode, kernel mode
Targets	Live debugging only
Platforms	All

Additional Information

For more information about and examples of how to use breakpoints, other breakpoint commands and methods of controlling breakpoints, and how to set breakpoints in user space from a kernel debugger, see Using Breakpoints. For more information about conditional breakpoints, see Setting a Conditional Breakpoint.

Remarks

The **bp**, **bu**, and **bm** commands set new breakpoints, but they have different characteristics:

- The **bp (Set Breakpoint)** command sets a new breakpoint at the *address* of the breakpoint location that is specified in the command. If the debugger cannot resolve the address expression of the breakpoint location when the breakpoint is set, the **bp** breakpoint is automatically converted to a **bu** breakpoint. Use a **bp** command to create a breakpoint that is no longer active if the module is unloaded.
- The **bu (Set Unresolved Breakpoint)** command sets a *deferred* or *unresolved* breakpoint. A **bu** breakpoint is set on a symbolic reference to the breakpoint location that is specified in the command (not on an address) and is activated whenever the module with the reference is resolved. For more information about these breakpoints, see Unresolved Breakpoints (bu Breakpoints).
- The **bm (Set Symbol Breakpoint)** command sets a new breakpoint on symbols that match a specified pattern. This command can create more than one breakpoint. By default, after the pattern is matched, **bm** breakpoints are the same as **bu** breakpoints. That is, **bm** breakpoints are deferred breakpoints that are set on a symbolic reference. However, a bm /d command creates one or more **bp** breakpoints. Each breakpoint is set on the address of a matched location and does not track module state.

If you are not sure what command was used to set an existing breakpoint, use **.bpcmds (Display Breakpoint Commands)** to list all breakpoints along with the commands that were used to create them.

There are three primary differences between **bp** breakpoints and **bu** breakpoints:

- A **bp** breakpoint location is always converted to an address. If a module change moves the code at which a **bp** breakpoint was set, the breakpoint remains at the same address. On the other hand, a **bu** breakpoint remains associated with the symbolic value (typically a symbol plus an offset) that was used, and it tracks this symbolic location even if its address changes.
- If a **bp** breakpoint address is found in a loaded module, and if that module is later unloaded, the breakpoint is removed from the breakpoint list. On the other hand, **bu** breakpoints persist after repeated unloads and loads.
- Breakpoints that you set with **bp** are not saved in WinDbg workspaces. Breakpoints that are set with **bu** are saved in workspaces.

The **bm** command is useful when you want to use wildcard characters in the symbol pattern for a breakpoint. The **bm** *SymbolPattern* syntax is equivalent to using **x SymbolPattern** and then using **bu** on each result. For example, to set breakpoints on all of the symbols in the *Myprogram* module that begin with the string "mem," use the following command.

Example

```
0:000> bm myprogram!mem*
  4: 0040d070 MyProgram!memcpy
 5: 0040c560 MyProgram!memmove
  6: 00408960 MyProgram!memset
```

Because the **bm** command sets software breakpoints (not processor breakpoints), it automatically excludes data location when it sets breakpoints to avoid corrupting the data.

It is possible to specify a data address rather than a program address when using the **bp** or bm /a commands. However, even if a data location is specified, these commands create software breakpoints, not processor breakpoints. If a software breakpoint is placed in program data instead of executable code, it can

lead to data corruption. Therefore you should use these commands in a data location only if you are certain that the memory stored in that location will be used as executable code and not as program data. Otherwise, you should use the **ba (Break on Access)** command instead. For more details, see Processor Breakpoints (ba Breakpoints).

For details on how to set a breakpoint on a location specified by a more complicated syntax, such as a member of a C++ public class, or an arbitrary text string containing otherwise restricted characters, see Breakpoint Syntax.

If a single logical source line spans multiple physical lines, the breakpoint is set on the last physical line of the statement or call. If the debugger cannot set a breakpoint at the requested position, it puts the breakpoint in the next allowed position.

If you specify *Thread*, breakpoints are set on the specified threads. For example, the **~*bp** command sets breakpoints on all threads, **~#bp** sets a breakpoint on the thread that causes the current exception, and **~123bp** sets a breakpoint on thread 123. The **~bp** and **~.bp** commands both set a breakpoint on the current thread.

When you are debugging a multiprocessor system in kernel mode, breakpoints that you set by using **bp** or **ba (Break on Access)** apply to all processors. For example, if the current processor is 3 and you type **bp MemoryAddress** to put a breakpoint at **MemoryAddress**. Any processor that is executing at that address (not only processor 3) causes a breakpoint trap.

The **bp**, **bu**, and **bm** commands set software breakpoints by replacing the processor instruction with a break instruction. To debug read-only code or code that cannot be changed, use a ba e command, where **e** represents execute-only access.

The following command sets a breakpoint 12 bytes past the beginning of the function **MyTest**. This breakpoint is ignored for the first six passes through the code, but execution stops on the seventh pass through the code.

```
0:000> bp MyTest+0xb 7
```

The following command sets a breakpoint at **RtlRaiseException**, displays the **eax** register, displays the value of the symbol **MyVar**, and continues.

```
kd> bp ntdll!RtlRaiseException "r eax; dt MyVar; g"
```

The following two **bm** commands set three breakpoints. When the commands are executed, the displayed result does not distinguish between breakpoints created with the **/d** switch and those created without it. The **.bpcmds (Display Breakpoint Commands)** can be used to distinguish between these two types. If the breakpoint was created by **bm** without the **/d** switch, the **.bpcmds** display indicates the breakpoint type as **bu**, followed by the evaluated symbol enclosed in the **@!""** token (which indicates it is a literal symbol and not a numeric expression or register). If the breakpoint was created by **bm** with the **/d** switch, the **.bpcmds** display indicates the breakpoint type as **bp**.

```
0:000> bm myprog!openf*
  0: 00421200 @!"myprog!openFile"
  1: 00427800 @!"myprog!openFilter"

0:000> bm /d myprog!closef*
  2: 00421600 @!"myprog!closeFile"

0:000> .bpcmds
bu0 @!"myprog!openFile";
bu1 @!"myprog!openFilter";
bp2 0x00421600 ;
```

br (Breakpoint Renumber)

12/5/2018 • 2 minutes to read • Edit Online

The **br** command renumbers one or more breakpoints.

```
br OldID NewID [OldID2 NewID2 ...]
```

Parameters

OldID
Specifies the current ID number of the breakpoint.

NewID
Specifies a new number that becomes the ID of the breakpoint.

Environment

Modes	User mode, kernel mode
Targets	Live debugging only
Platforms	All

Additional Information

For more information about and examples of how to use breakpoints, other breakpoint commands and methods of controlling breakpoints, and how to set breakpoints in user space from a kernel debugger, see Using Breakpoints. For more information about conditional breakpoints, see Setting a Conditional Breakpoint.

Remarks

You can use the **br** command to renumber any number of breakpoints at the same time. For each breakpoint, list the old ID and the new ID, in that order, as parameters to **br**.

If there is already a breakpoint with an ID equal to *NewID*, the command fails and an error message is displayed.

bs (Update Breakpoint Command)

12/5/2018 • 2 minutes to read • Edit Online

The **bs** command changes the command executed when the specified breakpoint is encountered.

```
bs ID ["CommandString"]
```

Parameters

ID
Specifies the ID number of the breakpoint.

CommandString
Specifies the new list of commands to be executed every time that the breakpoint is encountered. You must enclose the *CommandString* parameter in quotation marks. Use semicolons to separate multiple commands.

Debugger commands in *CommandString* can include parameters. You can use standard C-control characters (such as **\n** and **\"**). Semicolons that are contained in second-level quotation marks (**\"**) are interpreted as part of the embedded quoted string.

The *CommandString* commands are executed only if the breakpoint is reached while the application is executing in response to a **g (Go)** command. The commands are not executed if you are stepping through the code or tracing past this point.

Any command that resumes program execution after a breakpoint (such as **g** or **t**) ends the execution of the command list.

Environment

Modes	User mode, kernel mode
Targets	Live debugging only
Platforms	All

Additional Information

For more information about and examples of how to use breakpoints, other breakpoint commands and methods of controlling breakpoints, and how to set breakpoints in user space from a kernel debugger, see Using Breakpoints. For more information about conditional breakpoints, see Setting a Conditional Breakpoint.

Remarks

If the *CommandString* is not specified, any commands already set on the breakpoint are removed.

bsc (Update Conditional Breakpoint)

12/5/2018 • 2 minutes to read • Edit Online

The **bsc** command changes the condition under which a breakpoint occurs or changes the command executed when the specified conditional breakpoint is encountered.

```
bsc ID Condition ["CommandString"]
```

Parameters

ID
Specifies the ID number of the breakpoint.

Condition
Specifies the condition under which the breakpoint should be triggered.

CommandString
Specifies the new list of commands to be executed every time that the breakpoint is encountered. You must enclose the *CommandString* parameter in quotation marks. Use semicolons to separate multiple commands.

Debugger commands in *CommandString* can include parameters. You can use standard C-control characters (such as **\n** and **\"**). Semicolons that are contained in second-level quotation marks (**\"**) are interpreted as part of the embedded quoted string.

The CommandString commands are executed only if the breakpoint is reached while the application is executing in response to a **g (Go)** command. The commands are not executed if you are stepping through the code or tracing past this point.

Any command that resumes program execution after a breakpoint (such as **g** or **t**) ends the execution of the command list.

Environment

Modes	User mode, kernel mode
Targets	Live debugging only
Platforms	All

Additional Information

For more information about and examples of how to use breakpoints, other breakpoint commands and methods of controlling breakpoints, and how to set breakpoints in user space from a kernel debugger, see Using Breakpoints. For more information about conditional breakpoints, see Setting a Conditional Breakpoint.

Remarks

If the *CommandString* is not specified, any commands already set on the breakpoint are removed.

The same effect can be achieved by using the **bs (Update Breakpoint Command)** command with the following

syntax:

```
bs ID "j Condition 'CommandString'; 'gc'"
```

c (Compare Memory)

12/5/2018 • 2 minutes to read • Edit Online

The **c** command compares the values held in two memory areas.

```
c Range Address
```

Parameters

Range
The first of the two memory ranges to be compared. For more syntax details, see Address and Address Range Syntax.

Address
The starting address of the second memory range to be compared. The size of this range will be the same as that specified for the first range. For more syntax details, see Address and Address Range Syntax.

Environment

Modes	user mode, kernel mode
Targets	live, crash dump
Platforms	all

Additional Information

For an overview of memory manipulation and a description of other memory-related commands, see Reading and Writing Memory.

Remarks

If the two areas are not identical, the debugger will display all memory addresses in the first range where they do not agree.

As an example, consider the following code:

```
void main()
{
    char rgBuf1[100];
    char rgBuf2[100];

    memset(rgBuf1, 0xCC, sizeof(rgBuf1));
    memset(rgBuf2, 0xCC, sizeof(rgBuf2));

    rgBuf1[42] = 0xFF;
}
```

To compare **rgBuf1** and **rgBuf2**, use either of the following commands:

```
0:000> c rgBuf1 (rgBuf1+0n100) rgBuf2

0:000> c rgBuf1 L 0n100 rgBuf2
```

d, da, db, dc, dd, dD, df, dp, dq, du, dw (Display Memory)

7/9/2019 • 4 minutes to read • Edit Online

The **d*** commands display the contents of memory in the given range.

```
d{a|b|c|d|D|f|p|q|u|w|W} [Options] [Range]
dy{b|d} [Options] [Range]
d [Options] [Range]
```

Parameters

Options
Specifies one or more display options. Any of the following options can be included, except that no more than one **/p*** option can be indicated:

/c*Width*
Specifies the number of columns to use in the display. If this is omitted, the default number of columns depends on the display type.

/p
(Kernel-mode only) Uses physical memory addresses for the display. The range specified by *Range* will be taken from physical memory rather than virtual memory.

/p[c]
(Kernel-mode only) Same as **/p**, except that cached memory will be read. The brackets around **c** must be included.

/p[uc]
(Kernel-mode only) Same as **/p**, except that uncached memory will be read. The brackets around **uc** must be included.

/p[wc]
(Kernel-mode only) Same as **/p**, except that write-combined memory will be read. The brackets around **wc** must be included.

Range
Specifies the memory area to display. For more syntax details, see Address and Address Range Syntax. If you omit *Range*, the command will display memory starting at the ending location of the last display command. If *Range* is omitted and no previous display command has been used, the display begins at the current instruction pointer.

Environment

Modes	user mode, kernel mode
Targets	live, crash dump
Platforms	all

Additional Information

For an overview of memory manipulation and a description of other memory-related commands, see Reading and Writing Memory.

Remarks

Each line displayed will include the address of the first byte in the line followed by the contents of memory at that and following locations.

If you omit *Range*, the command will display memory starting at the ending location of the last display command. This allows you to continuously scan through memory.

This command exists in the following forms. The second characters of the **dd**, **dD**, **dw**, and **dW** commands are case-sensitive, as are the third characters of the **dyb** and **dyd** commands.

COMMAND	DISPLAY
d	This displays data in the same format as the most recent d command. If no previous d command has been issued, d has the same effect as db. Notice that d repeats the most recent command that began with d. This includes dda, ddp, ddu, dpa, dpp, dpu, dqa, dqp, dqu, dds, dps, dqs, ds, dS, dg, dl, dt, and dv, as well as the display commands on this page. If the parameters given after d are not appropriate, errors may result.
da	ASCII characters. Each line displays up to 48 characters. The display continues until the first null byte or until all characters in range have been displayed. All nonprintable characters, such as carriage returns and line feeds, are displayed as periods (.).
db	Byte values and ASCII characters. Each display line shows the address of the first byte in the line, followed by up to 16 hexadecimal byte values. The byte values are immediately followed by the corresponding ASCII values. The eighth and ninth hexadecimal values are separated by a hyphen (-). All nonprintable characters, such as carriage returns and line feeds, are displayed as periods (.). The default count is 128 bytes.
dc	Double-word values (4 bytes) and ASCII characters. Each display line shows the address of the first word in the line and up to eight hexadecimal word values, as well as their ASCII equivalent. The default count is 32 DWORDs (128 bytes).
dd	Double-word values (4 bytes).The default count is 32 DWORDs (128 bytes).
dD	Double-precision floating-point numbers (8 bytes). The default count is 15 numbers (120 bytes).
df	Single-precision floating-point numbers (4 bytes). The default count is 16 numbers (64 bytes).

COMMAND	DISPLAY
dp	Pointer-sized values. This command is equivalent to dd or dq, depending on whether the target computer's processor architecture is 32-bit or 64-bit, respectively. The default count is 32 DWORDs or 16 quad-words (128 bytes).
dq	Quad-word values (8 bytes). The default count is 16 quad-words (128 bytes).
du	Unicode characters. Each line displays up to 48 characters. The display continues until the first null byte or until all characters in range have been displayed. All nonprintable characters, such as carriage returns and line feeds, are displayed as periods (.).
dw	Word values (2 bytes). Each display line shows the address of the first word in the line and up to eight hexadecimal word values. The default count is 64 words (128 bytes).
dW	Word values (2 bytes) and ASCII characters. Each display line shows the address of the first word in the line and up to eight hexadecimal word values. The default count is 64 words (128 bytes).
dyb	Binary values and byte values. The default count is 32 bytes.
dyd	Binary values and double-word values (4 bytes). The default count is 8 DWORDs (32 bytes).

If you attempt to display an invalid address, its contents are shown as question marks (**?**).

dda, ddp, ddu, dpa, dpp, dpu, dqa, dqp, dqu (Display Referenced Memory)

7/9/2019 • 2 minutes to read • Edit Online

The **dda**, **ddp**, **ddu**, **dpa**, **dpp**, **dpu**, **dqa**, **dqp**, and **dqu** commands display the pointer at the specified location, dereference that pointer, and then display the memory at the resulting location in a variety of formats.

```
ddp [Options] [Range]
dqp [Options] [Range]
dpp [Options] [Range]
dda [Options] [Range]
dqa [Options] [Range]
dpa [Options] [Range]
ddu [Options] [Range]
dqu [Options] [Range]
dpu [Options] [Range]
```

Parameters

Options
Specifies one or more display options. Any of the following options can be included, except that no more than one **/p*** option can be indicated:

/c*Width*
Specifies the number of columns to use in the display. If this is omitted, the default number of columns depends on the display type. Because of the way pointers are displayed by these commands, it is usually best to use the default of only one data column.

/p
(Kernel-mode only) Uses physical memory addresses for the display. The range specified by *Range* will be taken from physical memory rather than virtual memory.

/p[c]
(Kernel-mode only) Same as **/p**, except that cached memory will be read. The brackets around **c** must be included.

/p[uc]
(Kernel-mode only) Same as **/p**, except that uncached memory will be read. The brackets around **uc** must be included.

/p[wc]
(Kernel-mode only) Same as **/p**, except that write-combined memory will be read. The brackets around **wc** must be included.

Range
Specifies the memory area to display. For more syntax details, see Address and Address Range Syntax. If you omit *Range*, the command will display memory starting at the ending location of the last display command. If *Range* is omitted and no previous display command has been used, the display begins at the current instruction pointer. If a simple address is given, the default range length is 128 bytes.

Environment

Modes	user mode, kernel mode
Targets	live, crash dump
Platforms	all

Additional Information

For an overview of memory manipulation and a description of other memory-related commands, see Reading and Writing Memory.

Remarks

The second and third characters of this command are case-sensitive.

The second character of this command determines the pointer size used:

COMMAND	DISPLAY
dd	32-bit pointers used
dq	64-bit pointers used
dp*	Standard pointer sizes used: 32-bit or 64-bit, depending on the target's processor architecture

The third character of this command determines how the dereferenced memory is displayed:

COMMAND	DISPLAY
dp	Displays the contents of the memory referenced by the pointer in DWORD or QWORD format, depending on the pointer size of the target's processor architecture. If this value matches any known symbol, this symbol is displayed as well.
da	Displays the contents of the memory referenced by the pointer in ASCII character format.
d*u	Displays the contents of the memory referenced by the pointer in Unicode character format.

If line number information has been enabled, source file names and line numbers will be displayed when available.

dds, dps, dqs (Display Words and Symbols)

12/5/2018 • 2 minutes to read • Edit Online

The **dds**, **dps**, and **dqs** commands display the contents of memory in the given range. This memory is assumed to be a series of addresses in the symbol table. The corresponding symbols are displayed as well.

```
dds [Options] [Range]
dqs [Options] [Range]
dps [Options] [Range]
```

Parameters

Options
Specifies one or more display options. Any of the following options can be included, except that no more than one **/p*** option can be indicated:

/c *Width*
Specifies the number of columns to use in the display. If this is omitted, the default number of columns depends on the display type. Because of the way symbols are displayed by these commands, it is usually best to use the default of only one data column.

/p
(Kernel-mode only) Uses physical memory addresses for the display. The range specified by *Range* will be taken from physical memory rather than virtual memory.

/p[c]
(Kernel-mode only) Same as **/p**, except that cached memory will be read. The brackets around **c** must be included.

/p[uc]
(Kernel-mode only) Same as **/p**, except that uncached memory will be read. The brackets around **uc** must be included.

/p[wc]
(Kernel-mode only) Same as **/p**, except that write-combined memory will be read. The brackets around **wc** must be included.

Range
Specifies the memory area to display. For more syntax details, see Address and Address Range Syntax. If you omit *Range*, the command will display memory starting at the ending location of the last display command. If *Range* is omitted and no previous display command has been used, the display begins at the current instruction pointer. If a simple address is given, the default range length is 128 bytes.

Environment

Modes	user mode, kernel mode
Targets	live, crash dump
Platforms	all

Additional Information

For an overview of memory manipulation and a description of other memory-related commands, see Reading and Writing Memory.

Remarks

The second character of **dds** is case-sensitive. The third character of all these commands is case-sensitive.

The **dds** command displays double-word (4 byte) values like the **dd** command. The **dqs** command displays quad-word (8 byte) values like the **dq** command. The **dps** command displays pointer-sized values (4 byte or 8 byte, depending on the target computer's architecture) like the **dp** command.

Each of these words is treated as an address in the symbol table. The corresponding symbol information is displayed for each word.

If line number information has been enabled, source file names and line numbers will be displayed when available.

dg (Display Selector)

12/5/2018 • 2 minutes to read • Edit Online

The **dg** command shows the segment descriptor for the specified selector.

```
dg FirstSelector [LastSelector]
```

Parameters

FirstSelector
Specifies the hexadecimal selector value of the first selector to be displayed.

LastSelector
Specifies the hexadecimal selector value of the last selector to be displayed. If this is omitted, only one selector will be displayed.

Environment

Modes	user mode, kernel mode
Targets	live, crash dump
Platforms	x86, Itanium

Remarks

No more than 256 selectors can be displayed by this command.

Common selector values are:

ID	DECIMAL	HEX
KGDT_NULL	0	0x00
KGDT_R0_CODE	8	0x08
KGDT_R0_DATA	16	0x10
KGDT_R3_CODE	24	0x18
KGDT_R3_DATA	32	0x20
KGDT_TSS	40	0x28

ID	DECIMAL	HEX
KGDT_R0_PCR	48	0x30
KGDT_R3_TEB	56	0x38
KGDT_VDM_TILE	64	0x40
KGDT_LDT	72	0x48
KGDT_DF_TSS	80	0x50
KGDT_NMI_TSS	88	0x58

dl (Display Linked List)

12/5/2018 • 2 minutes to read • Edit Online

The **dl** command displays a LIST_ENTRY or SINGLE_LIST_ENTRY linked list.

```
dl[b] Address MaxCount Size
```

Parameters

b

If this is included, the list is dumped in reverse order. (In other words, the debugger follows the **Blink**s instead of the **Flink**s.) This cannot be used with a SINGLE_LIST_ENTRY.

Address

The starting address of the list. For more syntax details, see Address and Address Range Syntax.

MaxCount

Maximum number of elements to dump.

Size

Size of each element. This is the number of consecutive ULONG_PTRs that will be displayed for each element in the list.

Environment

Modes	user mode, kernel mode
Targets	live, crash dump
Platforms	all

Additional Information

For an overview of memory manipulation and a description of other memory-related commands, see Reading and Writing Memory.

Remarks

This list must be a LIST_ENTRY or SINGLE_LIST_ENTRY structure. If this is embedded in a larger structure, be sure that *Address* points to the linked list structure and not to the beginning of the outer structure.

The display begins with *Address*. Therefore, if you are supplying the address of a pointer that points to the beginning of the list, you should disregard the first element printed.

The *Address*, *MaxCount*, and *Size* parameters are in the current default radix. You can use the **n (Set Number Base)** command or the **0x** prefix to change the radix.

If the list loops back on itself, the dump will stop. If a null pointer is encountered, the dump will stop.

If you want to execute some command for each element of the list, use the **!list** extension.

ds, dS (Display String)

12/5/2018 • 2 minutes to read • Edit Online

The **ds** and **dS** commands display a STRING, ANSI_STRING, or UNICODE_STRING *structures*.

These commands do not display null-delimited character strings, but rather string structures.

If you have the address of the characters of a Unicode string then use the du command instead. Use the da command to display ASCII characters. For more information, see d, da, db, dc, dd, dD, df, dp, dq, du, dw (Display Memory).

```
d{s|S} [/c Width] [Address]
```

Parameters

s
Specifies that a STRING or ANSI_STRING structure is to be displayed. (This **s** is case-sensitive.)

S
Specifies that a UNICODE_STRING structure is to be displayed. (This **S** is case-sensitive.)

***/c** Width*
Specifies the number of characters to display on each line. This number includes null characters, which will not be visible.

Address
The memory address where the where the UNICODE_STRING structure is stored.

For more syntax details, see Address and Address Range Syntax. If omitted, the last address used in a display command is assumed.

Environment

Modes	user mode, kernel mode
Targets	live, crash dump
Platforms	all

Additional Information

For an overview of memory manipulation and a description of other memory-related commands, see Reading and Writing Memory.

Remarks

If you want to display Unicode strings in the Locals window or Watch window of WinDbg, you need to use the **.enable_unicode (Enable Unicode Display)** command first.

dt (Display Type)

9/25/2019 • 11 minutes to read • Edit Online

The **dt** command displays information about a local variable, global variable or data type. This can display information about simple data types, as well as structures and unions.

User-Mode Syntax

```
dt [-DisplayOpts] [-SearchOpts] [module!]Name [[-SearchOpts] Field] [Address] [-l List]
dt [-DisplayOpts] Address [-l List]
dt -h
```

Kernel-Mode Syntax

```
[Processor] dt [-DisplayOpts] [-SearchOpts] [module!]Name [[-SearchOpts] Field] [Address] [-l List]
dt [-DisplayOpts] Address [-l List]
dt -h
```

Parameters

Processor
Specifies the processor that is running the process containing the information needed. For more information, see Multiprocessor Syntax. Processors can only be specified in kernel mode.

DisplayOpts
Specifies one or more of the options given in the following table. These options are preceded by a hyphen.

OPTION	DESCRIPTION
-a[*quantity*]	Show each array element on a new line, with its index. A total of *quantity* elements will be displayed. There must be no space between the **a** and the *quantity*. If **-a** is not followed by a digit, all items in the array are shown. The **-a**[*quantity*] switch should appear immediately before each type name or field name that you want displayed in this manner.
-b	Display blocks recursively. If a displayed structure contains substructures, it is expanded recursively to arbitrary depths and displayed in full. Pointers are expanded only if they are in the original structure, not in substructures.
-c	Compact output. All fields are displayed on one line, if possible. (When used with the **-a** switch, each array element takes one line rather than being formatted as a several-line block.)

OPTION	DESCRIPTION
-d	When used with a *Name* that is ended with an asterisk, display verbose output for all types that begin with *Name*. If *Name* does not end with an asterisk, display verbose output.
-e	Forces **dt** to enumerate types. This option is only needed if **dt** is mistakenly interpreting the *Name* value as an instance rather than as a type.
-i	Do not indent the subtypes.
-o	Omit offset values of the structure fields.
-p	*Address* is a physical address, rather than a virtual address.
-r[*depth*]	Recursively dumps the subtype fields. If *depth* is given, this recursion will stop after *depth* levels. The *depth* must be a digit between 1 and 9, and there must be no space between the **r** and the *depth*. The **-r**[*depth*] switch should appear immediately before the address.
-s *size*	Enumerate only those types whose size in bytes equals the value of *size*. The **-s** option is only useful when types are being enumerated. When **-s** is specified, **-e** is always implied as well.
-t	Enumerate types only.
-v	Verbose output. This gives additional information such as the total size of a structure and the number of its elements. When this is used along with the **-y** search option, all symbols are displayed, even those with no associated type information.

SearchOpts

Specifies one or more of the options given in the following table. These options are preceded by a hyphen.

OPTION	DESCRIPTION
-n	This indicates that the next parameter is a name. This should be used if the next item consists entirely of hexadecimal characters, because it will otherwise be taken as an address.

OPTION	DESCRIPTION
-y	This indicates that the next parameter is the beginning of the name, not necessarily the entire name. When **-y** is included, all matches are listed, followed by detailed information on the first match in the list. If **-y** is not included, only exact matches will be displayed.

module
An optional parameter specifying the module that defines this structure. If there is a local variable or type with the same name as a global variable or type, you should include *module* to specify that you mean the global variable. Otherwise, the **dt** command will display the local variable, even if the local variable is a case-insensitive match and the global variable is a case-sensitive match.

Name
Specifies the name of a type or global variable. If *Name* ends with an asterisk (*), a list of all matches is displayed. Thus, **dt A*** will list all data types, globals, and statics beginning with "A", but will not display the actual instances of these types. (If the **-v** display option is used at the same time, all symbols will be displayed -- not just those with associated type information.) You can also replace *Name* with a period (**.**) to signify that you want to repeat the most recently used value of *Name*.

If *Name* contains a space, it should be enclosed in parentheses.

Field
Specifies the field(s) to be displayed. If *Field* is omitted, all fields are displayed. If *Field* is followed by a period (**.**), the first-level subfields of this field will be displayed as well. If *Field* is followed with a series of periods, the subfields will be displayed to a depth equal to the number of periods. Any field name followed by a period will be treated as a prefix match, as if the **-y** search option was used. If *Field* is followed by an asterisk (*), it is treated as only the beginning of the field, not necessarily the entire field, and all matching fields are displayed.

Address
Specifies the address of the structure to be displayed. If *Name* is omitted, *Address* must be included and must specify the address of a global variable. *Address* is taken to be a virtual address unless otherwise specified. Use the **-p** option to specify a physical address. Use an "at" sign (**@**) to specify a register (for example, **@eax**).

List
Specifies the field name that links a linked list. The *Address* parameter must be included.

Environment

Modes	user mode, kernel mode
Targets	live, crash dump
Platforms	all

Additional Information

For an overview of memory manipulation and a description of other memory-related commands, see Reading and Writing Memory.

Remarks

The **dt** command output will always display signed numbers in base 10, and unsigned numbers in hexadecimal.

All parameters of **dt** that allow symbol values also allow string wildcards. See String Wildcard Syntax for details.

The **-y** and **-n** options can precede any *Name* or *Field*. The **-y** option allows you to specify the beginning of the type or structure name. For example, **dt -y ALLEN** will display data about the type **ALLENTOWN**. However, you could not display the type **ALLENTOWN** with **dt -y A**. Instead, you would have to use **dt -ny A**, because **A** is a valid hexadecimal value and would be interpreted as an address without the **-n** option.

If *Name* indicates a structure, all fields will be displayed (for example, **dt myStruct**). If you only want one specific field, you can do **dt myStruct myField**. This displays the member that C would call **myStruct.myField**. However, note that the command **dt myStruct myField1 myField2** displays **myStruct.myField1** and **myStruct.myField2**. It does not display **myStruct.myField1.myField2**.

If a structure name or field is followed by a subscript, this specifies a single instance of an array. For example, **dt myStruct myFieldArray[3]** will display the fourth element of the array in question. But if a type name is followed by a subscript, this specifies an entire array. For example, **dt CHAR[8] myPtr** will display an eight-character string. The subscript is always taken as decimal regardless of the current radix; an **0x** prefix will cause an error.

Because the command uses type information from the *.pdb* file, it can freely be used to debug any CPU platform.

The type information used by **dt** includes all type names created with **typedef**, including all the Windows-defined types. For example, **unsigned long** and **char** are not valid type names, but **ULONG** and **CHAR** are. See the Microsoft Windows SDK for a full list of all Windows type names.

All types created by **typedefs** within your own code will be present, as long as they have actually been used in your program. However, types that are defined in your headers but never actually used will not be stored in the .pdb symbol files and will not be accessible to the debugger. To make such a type available to the debugger, use it as the *input* of a **typedef** statement. For example, if the following appears in your code, the structure MY_DATA will be stored in the .pdb symbol file and can be displayed by the **dt** command:

```
typedef struct _MY_DATA {
    . . .
    } MY_DATA;
typedef  MY_DATA *PMY_DATA;
```

On the other hand, the following code would not suffice because both MY_DATA and PMY_DATA are defined by the initial **typedef** and, therefore, MY_DATA has not itself been used as the input of any **typedef** statement:

```
typedef struct _MY_DATA {
    . . .
    } MY_DATA, *PMY_DATA;
```

In any event, type information is included only in a full symbol file, not a symbol file that has been stripped of all private symbol information. For more information, see Public and Private Symbols.

If you want to display unicode strings, you need to use the **.enable_unicode (Enable Unicode Display)**

command first. You can control the display of long integers with the **.enable_long_status (Enable Long Integer Display)** command.

In the following example, **dt** displays a global variable:

```
0:000> dt mt1
   +0x000 a                : 10
   +0x004 b                : 98 'b'
   +0x006 c                : 0xdd
   +0x008 d                : 0xabcd
   +0x00c gn               : [6] 0x1
   +0x024 ex               : 0x0
```

In the following example, **dt** displays the array field **gn**:

```
0:000> dt mt1 -a gn
   +0x00c gn :
    [00] 0x1
    [01] 0x2
    [02] 0x3
    [03] 0x4
    [04] 0x5
    [05] 0x6
```

The following command displays some subfields of a variable:

```
0:000> dt mcl1 m_t1 dpo
   +0x010 dpo  : DEEP_ONE
   +0x070 m_t1 : MYTYPE1
```

The following command displays the subfields of the field **m_t1**. Because the period automatically causes prefix matching, this will also display subfields of any field that begins with "m_t1":

```
0:000> dt mcl1 m_t1.
   +0x070 m_t1  :
      +0x000 a     : 0
      +0x004 b     : 0 '
      +0x006 c     : 0x0
      +0x008 d     : 0x0
      +0x00c gn    : [6] 0x0
      +0x024 ex    : 0x0
```

You could repeat this to any depth. For example, the command **dt mcl1 a..c.** would display all fields to depth four, such that the first field name began with **a** and the third field name began with **c**.

Here is a more detailed example of how subfields can be displayed. First, display the **Ldr** field:

```
0:000> dt nt!_PEB Ldr 7ffdf000
   +0x00c Ldr : 0x00191ea0
```

Now expand the pointer type field:

```
0:000> dt nt!_PEB Ldr Ldr. 7ffdf000
   +0x00c Ldr  : 0x00191ea0
      +0x000 Length : 0x28
      +0x004 Initialized : 0x1 '
      +0x008 SsHandle : (null)
      +0x00c InLoadOrderModuleList : _LIST_ENTRY [ 0x191ee0 - 0x192848 ]
      +0x014 InMemoryOrderModuleList : _LIST_ENTRY [ 0x191ee8 - 0x192850 ]
      +0x01c InInitializationOrderModuleList : _LIST_ENTRY [ 0x191f58 - 0x192858 ]
      +0x024 EntryInProgress : (null)
```

Now display the **CriticalSectionTimeout** field:

```
0:000> dt nt!_PEB CriticalSectionTimeout 7ffdf000
   +0x070 CriticalSectionTimeout : _LARGE_INTEGER 0xffffe86d`079b8000
```

Now expand the **CriticalSectionTimeout** structure subfields one level deep:

```
0:000> dt nt!_PEB CriticalSectionTimeout. 7ffdf000
   +0x070 CriticalSectionTimeout  :  0xffffe86d`079b8000
      +0x000 LowPart                 : 0x79b8000
      +0x004 HighPart                : -6035
      +0x000 u                       : __unnamed
      +0x000 QuadPart                : -25920000000000
```

Now expand the **CriticalSectionTimeout** structure subfields two levels deep:

```
0:000> dt nt!_PEB CriticalSectionTimeout.. 7ffdf000
   +0x070 CriticalSectionTimeout   :  0xffffe86d`079b8000
      +0x000 LowPart                  : 0x79b8000
      +0x004 HighPart                 : -6035
      +0x000 u                        :
         +0x000 LowPart                  : 0x79b8000
         +0x004 HighPart                 : -6035
      +0x000 QuadPart                 : -25920000000000
```

The following command displays an instance of the data type MYTYPE1 that is located at the address 0x0100297C:

```
0:000> dt 0x0100297c MYTYPE1
   +0x000 a                : 22
   +0x004 b                : 43 '+'
   +0x006 c                : 0x0
   +0x008 d                : 0x0
   +0x00c gn               : [6] 0x0
   +0x024 ex               : 0x0
```

The following command displays an array of 10 ULONGs at the address 0x01002BE0:

```
0:000> dt -ca10 ULONG 01002be0
[0] 0x1001098
[1] 0x1
[2] 0xdead
[3] 0x7d0
[4] 0x1
[5] 0xcd
[6] 0x0
[7] 0x0
[8] 0x0
[9] 0x0
```

The following command continues the previous display at a different address. Note that "ULONG" does not need to be re-entered:

```
0:000> dt -ca4 . 01002d00
Using sym ULONG
[0] 0x12
[1] 0x4ac
[2] 0xbadfeed
[3] 0x2
```

Here are some examples of type display. The following command displays all types and globals beginning with the string "MY" in the module *thismodule*. Those prefixed with an address are actual instances; those without addresses are type definitions:

```
0:000> dt thismodule!MY*
010029b8  thismodule!myglobal1
01002990  thismodule!myglobal2
          thismodule!MYCLASS1
          thismodule!MYCLASS2
          thismodule!MYCLASS3
          thismodule!MYTYPE3::u
          thismodule!MYTYPE1
          thismodule!MYTYPE3
          thismodule!MYTYPE3
          thismodule!MYFLAGS
```

When performing type display, the **-v** option can be used to display the size of each item. The **-s** *size* option can be used to only enumerate items of a specific size. Again, those prefixed with an address are actual instances; those without addresses are type definitions:

```
0:001> dt -s 2 -v thismodule!*
Enumerating symbols matching thismodule!*, Size = 0x2
Address   Size Symbol
           002 thismodule!wchar_t
           002 thismodule!WORD
           002 thismodule!USHORT
           002 thismodule!SHORT
           002 thismodule!u_short
           002 thismodule!WCHAR
00427a34   002 thismodule!numberOfShips
00427a32   002 thismodule!numberOfPlanes
00427a30   002 thismodule!totalNumberOfItems
```

Here is an example of the **-b** option. The structure is expanded and the **OwnerThreads** array within the structure is expanded, but the **Flink** and **Blink** list pointers are not followed:

```
kd> dt nt!_ERESOURCE -b 0x8154f040
   +0x000 SystemResourcesList :  [ 0x815bb388 - 0x816cd478 ]
      +0x000 Flink            : 0x815bb388
      +0x004 Blink            : 0x816cd478
   +0x008 OwnerTable       : (null)
   +0x00c ActiveCount      : 1
   +0x00e Flag             : 8
   +0x010 SharedWaiters    : (null)
   +0x014 ExclusiveWaiters : (null)
   +0x018 OwnerThreads     :
    [00]
      +0x000 OwnerThread      : 0
      +0x004 OwnerCount       : 0
      +0x004 TableSize        : 0
    [01]
      +0x000 OwnerThread      : 0x8167f563
      +0x004 OwnerCount       : 1
      +0x004 TableSize        : 1
   +0x028 ContentionCount  : 0
   +0x02c NumberOfSharedWaiters : 0
   +0x02e NumberOfExclusiveWaiters : 0
   +0x030 Address          : (null)
   +0x030 CreatorBackTraceIndex : 0
   +0x034 SpinLock         : 0
```

Here is an example of **dt** in kernel mode. The following command produces results similar to **!process 0 0**:

```
kd> dt nt!_EPROCESS -l ActiveProcessLinks.Flink -y Ima -yoi Uni 814856f0
## ActiveProcessLinks.Flink at 0x814856f0

UniqueProcessId : 0x00000008
ImageFileName : [16] "System"

## ActiveProcessLinks.Flink at 0x8138a030

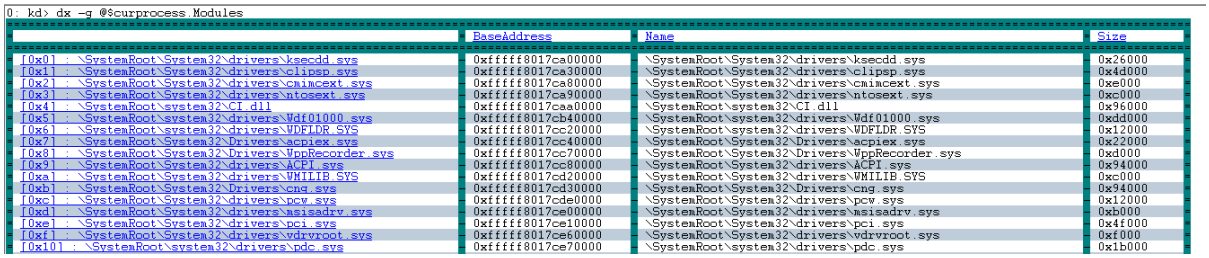
UniqueProcessId : 0x00000084
ImageFileName : [16] "smss.exe"

## ActiveProcessLinks.Flink at 0x81372368

UniqueProcessId : 0x000000a0
ImageFileName : [16] "csrss.exe"

## ActiveProcessLinks.Flink at 0x81369930

UniqueProcessId : 0x000000b4
ImageFileName : [16] "winlogon.exe"

....
```

If you want to execute a command for each element of the list, use the **!list** extension.

Finally, the **dt -h** command will display a short help text summarizing the **dt** syntax.

dtx (Display Type - Extended Debugger Object Model Information)

12/5/2018 • 2 minutes to read • Edit Online

The dtx command displays extended symbolic type information using the debugger object model. The dtx command is similar to the **dt (Display Type)** command.

```
dtx -DisplayOpts [Module!]Name Address
```

Parameters

DisplayOpts
Use the following optional flags to change how the output is displayed.

-a Displays array elements in a new line with its index.

-r [n] Recursively dump the subtypes (fields) up to *n* levels.

-h Displays command line help for the dtx command.

Module!
An optional parameter specifying the module that defines this structure, followed by the exclamation point. If there is a local variable or type with the same name as a global variable or type, you should include *module* name to specify the global variable.

Name
A type name or a global symbol.

Address
Memory address containing the type.

Environment

Modes	user mode, kernel mode
Targets	live, crash dump
Platforms	all

Additional Information

The following examples show how to use the dtx command.

Use the address and the name to display extended symbolic type information.

```
0: kd> dtx nt!_EPROCESS ffffb607560b56c0
(*((nt!_EPROCESS *)0xffffb607560b56c0))                [Type: _EPROCESS]
    [+0x000] Pcb              [Type: _KPROCESS]
    [+0x2d8] ProcessLock      [Type: _EX_PUSH_LOCK]
    [+0x2e0] RundownProtect   [Type: _EX_RUNDOWN_REF]
    [+0x2e8] UniqueProcessId  : 0x4 [Type: void *]
    [+0x2f0] ActiveProcessLinks [Type: _LIST_ENTRY]
```

Display additional information using the -r recursion option.

```
0: kd> dtx -r2 HdAudio!CAzMixertopoMiniport fffff806`d24992b8
(*((HdAudio!CAzMixertopoMiniport *)0xfffff806d24992b8))                [Type: CAzMixertopoMiniport]
    [+0x018] m_lRefCount      : -766760880 [Type: long]
    [+0x020] m_pUnknownOuter  : 0xfffff806d24dbc40 [Type: IUnknown *]
    [+0x028] m_FilterDesc     [Type: PCFILTER_DESCRIPTOR]
        [+0x000] Version          : 0xd24c2890 [Type: unsigned long]
        [+0x008] AutomationTable  : 0xfffff806d24c2780 [Type: PCAUTOMATION_TABLE *]
            [+0x000] PropertyItemSize : 0x245c8948 [Type: unsigned long]
            [+0x004] PropertyCount    : 0x6c894808 [Type: unsigned long]
            [+0x008] Properties       : 0x5718247489481024 [Type: PCPROPERTY_ITEM *]
            [+0x010] MethodItemSize   : 0x55415441 [Type: unsigned long]
            [+0x014] MethodCount      : 0x57415641 [Type: unsigned long]
            [+0x018] Methods          : 0x4ce4334540ec8348 [Type: PCMETHOD_ITEM *]
            [+0x020] EventItemSize    : 0x8b41f18b [Type: unsigned long]
            [+0x024] EventCount       : 0xd8b48f4 [Type: unsigned long]
            [+0x028] Events           : 0x7d2d8d4cfffdf854 [Type: PCEVENT_ITEM *]
            [+0x030] Reserved         : 0x66fffd79 [Type: unsigned long]
        [+0x010] PinSize          : 0xd24aa9b0 [Type: unsigned long]
        [+0x014] PinCount         : 0xfffff806 [Type: unsigned long]
        [+0x018] Pins             : 0xfffff806d24aa740 [Type: PCPIN_DESCRIPTOR *]
            [+0x000] MaxGlobalInstanceCount : 0x57555340 [Type: unsigned long]
            [+0x004] MaxFilterInstanceCount : 0x83485741 [Type: unsigned long]
            [+0x008] MinFilterInstanceCount : 0x8b4848ec [Type: unsigned long]
            [+0x010] AutomationTable  : 0xa5158b48ed33c000 [Type: PCAUTOMATION_TABLE *]
            [+0x018] KsPinDescriptor  [Type: KSPIN_DESCRIPTOR]
```

Tip: Use the **x (Examine Symbols)** command to display the address of an item of interest.

```
0: kd> x /d HdAudio!CazMixertopoMiniport*
...
fffff806`d24992b8 HdAudio!CAzMixertopoMiniport::`vftable' = <no type information>
...
```

See also

dt (Display Type)

dv (Display Local Variables)

12/5/2018 • 2 minutes to read • Edit Online

The **dv** command displays the names and values of all local variables in the current scope.

```
dv [Flags] [Pattern]
```

Parameters

Flags
Causes additional information to be displayed. Any of the following case-sensitive *Flags* can be included:

/f <addr>
Lets you specify an arbitrary function address so that you can see what parameters and locals exist for any code anywhere. It turns off the value display and implies **/V**. The **/f** flag must be the last flag. A parameter filter pattern can still be specified after it if the string is quoted.

/i
Causes the display to specify the kind of variable: local, global, parameter, function, or unknown.

/t
Causes the display to include the data type for each local variable.

/v
Causes the display to include the virtual memory address or register location of each local variable.

/V
Same as **/v**, and also includes the address of the local variable relative to the relevant register.

/a
Sorts the output by address, in ascending order.

/A
Sorts the output by address, in descending order.

/n
Sorts the output by name, in ascending order.

/N
Sorts the output by name, in descending order.

/z
Sorts the output by size, in ascending order.

/Z
Sorts the output by size, in descending order.

Pattern
Causes the command to only display local variables that match the specified *Pattern*. The pattern may contain a variety of wildcards and specifiers; see String Wildcard Syntax for details. If *Pattern* contains spaces, it must be enclosed in quotation marks. If *Pattern* is omitted, all local variables will be displayed.

Environment

Modes	user mode, kernel mode
Targets	live, crash dump
Platforms	all

Additional Information

For details on displaying and changing local variables and a description of other memory-related commands, see Reading and Writing Memory.

Remarks

In verbose mode, the addresses of the variables are displayed as well. (This can also be done with the **x (Examine Symbols)** command.)

Data structures and unfamiliar data types are not displayed in full; rather, their type name is displayed. To display the entire structure, or display a particular member of the structure, use the **dt (Display Type)** command.

The *local context* determines which set of local variables will be displayed. By default, this context matches the current position of the program counter. For information about how this can be changed, see Local Context.

dx (Display Debugger Object Model Expression)

6/25/2019 • 8 minutes to read • Edit Online

The **dx** command displays a C++ expression using the NatVis extension model. For more information about NatVis, see Create custom views of native objects.

```
dx [-g|-gc #][-c #][-n|-v]-r[#] Expression[,<FormatSpecifier> ]
dx [{-?}|{-h}]
```

Parameters

Expression
A C++ expression to be displayed.

-g
Display as a data grid objects which are iterable. Each iterated element is a row in the grid and each display child of those elements is a column. This allows you to view something such as an array of structs, where each array element is displayed in a row and each field of the struct is displayed in a column.

Left clicking a column name (where there is an available DML link) will sort by that column. If already sorted by that column, the sort order will be inverted.

Any object which is iterable will have a right click context menu item added via DML called 'Display as Grid'. Right clicking an object in the output window and selecting this will display the object in the grid view instead of the standard tree view.

A (+) displayed by a column name offers both a right click and left click behavior.

- Left click takes that column and explodes it into its own table. You see the original rows plus the children of the expanded column.
- Right click provides "Expand Into Grid" which takes the column and adds it back to the current table as right most columns.

-gc #
Display as a grid and restrict grid cell sizes to specified number of (#) characters.

-c #
Displays container continuation (skipping # elements of the container).This option is typically used in custom output automation scenarios and provides a "..." continuation element at the bottom of the listing.

-n
There are two ways that data can be rendered. Using the NatVis visualization (the default) or using the underlying native C/C++ structures. Specify the -n parameter to render the output using just the native C/C++ structures and not the NatVis visualizations.

-v
Display verbose information that includes methods and other non-typical objects.

-r#
Recursively display subtypes (fields) up to # levels. If # is not specified, a recursion level of one, is the default value.

[<,FormatSpecifier>]

Use any of the following format specifiers to modify the default rendering.

,x	Display ordinals in hexidecimal
,d	Display ordinals in decimal
,o	Display ordinals in octal
,b	Display ordinals in binary
,en	Display enums by name only (no value)
,c	Display as single character (not a string)
,s	Display 8-bit strings as ASCII quoted
,sb	Display 8-bit strings as ASCII unquoted
,s8	Display 8-bit strings as UTF-8 quoted
,s8b	Display 8-bit strings as UTF-8 unquoted
,su	Display 16-bit strings as UTF-16 quoted
,sub	Display 16-bit strings as UTF-16 unqouted
,!	Display objects in raw mode only (e.g.: no NatVis)
,#	Specify length of pointer/array/container as the literal value # (replace with numeric)
,[<expression>]	Specify length of pointer/array/container as the expression <expression>
,nd	Do not find the derived (runtype) type of the object. Display static value only

dx {**-?**}
Display command line help.

dx {**-h**}
Displays help for objects available in the debugger.

dx {**-id**}
Microsoft internal use only. Used to follow data model links in command output.

Command line usage example

The .dx settings command can be used to display information about the Debug Settings object. For more information about the debug settings objects, see **.settings** .

```
kd> dx -r1 Debugger.Settings
Debugger.Settings :
    Display          :
    EngineInitialization :
    Extensions       :
    Input            :
    Sources          :
    Symbols          :
    AutoSaveSettings : false
```

Use the -r1 recursion option to view the other Debugger objects - Sessions, Settings and State.

```
kd> dx -r1 Debugger
Debugger :
  Sessions :
  Settings :
  State    :
```

Specify the Debugger.Sessions object with the -r3 recursion option to travel further down the object chain.

```
kd> dx -r3 Debugger.Sessions
Debugger.Sessions :
  [0]              : Remote KD: KdSrv:Server=@{<Local>},Trans=@{1394:Channel=0}
    Processes :
      [0]              : <Unknown Image>
      [4]              : <Unknown Image>
      [304]            : smss.exe
      [388]            : csrss.exe
      [456]            : wininit.exe
      [468]            : csrss.exe
      [528]            : services.exe
      [536]            : lsass.exe
      [544]            : winlogon.exe
      [620]            : svchost.exe
       ...               ...
```

Add the x format specifier to display the ordinal values in hexadecimal.

```
kd> dx -r3 Debugger.Sessions,x
Debugger.Sessions,x :
  [0x0]            : Remote KD: KdSrv:Server=@{<Local>},Trans=@{1394:Channel=0}
    Processes :
      [0x0]            : <Unknown Image>
      [0x4]            : <Unknown Image>
      [0x130]          : smss.exe
      [0x184]          : csrss.exe
      [0x1c8]          : wininit.exe
      [0x1d4]          : csrss.exe
      [0x210]          : services.exe
      [0x218]          : lsass.exe
      [0x220]          : winlogon.exe
      [0x26c]          : svchost.exe
      [0x298]          : svchost.exe
      [0x308]          : dwm.exe
      [0x34c]          : nvvsvc.exe
      [0x37c]          : nvvsvc.exe
      [0x384]          : svchost.exe
       ...               ...
```

This example uses an active debug session to list the call stack of the first thread in the first process.

```
kd> dx -r1 Debugger.Sessions.First().Processes.First().Threads.First().Stack.Frames
Debugger.Sessions.First().Processes.First().Threads.First().Stack.Frames :
    [0x0]            : nt!RtlpBreakWithStatusInstruction
    [0x1]            : nt!KdCheckForDebugBreak + 0x7a006
    [0x2]            : nt!KiUpdateRunTime + 0x42
    [0x3]            : nt!KiUpdateTime + 0x129
    [0x4]            : nt!KeClockInterruptNotify + 0x1c3
    [0x5]            : hal!HalpTimerClockInterruptEpilogCommon + 0xa
    [0x6]            : hal!HalpTimerClockInterruptCommon + 0x3e
    [0x7]            : hal!HalpTimerClockInterrupt + 0x1cb
    [0x8]            : nt!KiIdleLoop + 0x1a
```

Use the -g option to display output as a data grid. Click on a column to sort.

```
kd> dx -g @$curprocess.Modules
```



```
0: kd> dx -g @$curprocess.Modules
```

	BaseAddress	Name	Size
[0x0] : \SystemRoot\System32\drivers\ksecdd.sys	0xfffff8017ca00000	\SystemRoot\System32\drivers\ksecdd.sys	0x26000
[0x1] : \SystemRoot\System32\drivers\clipsp.sys	0xfffff8017ca30000	\SystemRoot\System32\drivers\clipsp.sys	0x4d000
[0x2] : \SystemRoot\System32\drivers\cmimcext.sys	0xfffff8017ca80000	\SystemRoot\System32\drivers\cmimcext.sys	0xe000
[0x3] : \SystemRoot\System32\drivers\ntosext.sys	0xfffff8017ca90000	\SystemRoot\System32\drivers\ntosext.sys	0xc000
[0x4] : \SystemRoot\system32\CI.dll	0xfffff8017caa0000	\SystemRoot\system32\CI.dll	0x96000
[0x5] : \SystemRoot\system32\drivers\Wdf01000.sys	0xfffff8017cb40000	\SystemRoot\system32\drivers\Wdf01000.sys	0xdd000
[0x6] : \SystemRoot\system32\drivers\WDFLDR.SYS	0xfffff8017cc20000	\SystemRoot\system32\drivers\WDFLDR.SYS	0x12000
[0x7] : \SystemRoot\System32\Drivers\acpiex.sys	0xfffff8017cc40000	\SystemRoot\System32\Drivers\acpiex.sys	0x22000
[0x8] : \SystemRoot\System32\Drivers\WppRecorder.sys	0xfffff8017cc70000	\SystemRoot\System32\Drivers\WppRecorder.sys	0xd000
[0x9] : \SystemRoot\System32\drivers\ACPI.sys	0xfffff8017cc80000	\SystemRoot\System32\drivers\ACPI.sys	0x94000
[0xa] : \SystemRoot\System32\drivers\WMILIB.SYS	0xfffff8017cd20000	\SystemRoot\System32\drivers\WMILIB.SYS	0xc000
[0xb] : \SystemRoot\System32\Drivers\cng.sys	0xfffff8017cd30000	\SystemRoot\System32\Drivers\cng.sys	0x94000
[0xc] : \SystemRoot\System32\drivers\pcw.sys	0xfffff8017cde0000	\SystemRoot\System32\drivers\pcw.sys	0x12000
[0xd] : \SystemRoot\System32\drivers\msisadrv.sys	0xfffff8017ce00000	\SystemRoot\System32\drivers\msisadrv.sys	0xb000
[0xe] : \SystemRoot\System32\drivers\pci.sys	0xfffff8017ce10000	\SystemRoot\System32\drivers\pci.sys	0x4f000
[0xf] : \SystemRoot\System32\drivers\vdrvroot.sys	0xfffff8017ce60000	\SystemRoot\System32\drivers\vdrvroot.sys	0xf000
[0x10] : \SystemRoot\system32\drivers\pdc.sys	0xfffff8017ce70000	\SystemRoot\system32\drivers\pdc.sys	0x1b000

Use the -h option to display information about objects.

```
kd>  dx -h Debugger.State
Debugger.State   [State pertaining to the current execution of the debugger (e.g.: user variables)]
    DebuggerVariables [Debugger variables which are owned by the debugger and can be referenced by a
pseudo-register prefix of @$]
    PseudoRegisters   [Categorizied debugger managed pseudo-registers which can be referenced by a
pseudo-register prefix of @$]
    UserVariables     [User variables which are maintained by the debugger and can be referenced by a
pseudo-register prefix of @$]
```

Displaying TEB and PEB information using the Environment object

Use the Environment object to display TEB and PEB information associated with the thread and process.

To display the TEB associated with the current thread use this command.

```
0: kd> dx -r2 @$curthread.Environment
@$curthread.Environment
    EnvironmentBlock [Type: _TEB]
        [+0x000] NtTib            [Type: _NT_TIB]
        [+0x038] EnvironmentPointer : Unable to read memory at Address 0x38
        [+0x040] ClientId         [Type: _CLIENT_ID]
        [+0x050] ActiveRpcHandle  : Unable to read memory at Address 0x50
        [+0x058] ThreadLocalStoragePointer : Unable to read memory at Address 0x58
        [+0x060] ProcessEnvironmentBlock : Unable to read memory at Address 0x60
        [+0x068] LastErrorValue   : Unable to read memory at Address 0x68
        [+0x06c] CountOfOwnedCriticalSections : Unable to read memory at Address 0x6c
        [+0x070] CsrClientThread  : Unable to read memory at Address 0x70
        [+0x078] Win32ThreadInfo  : Unable to read memory at Address 0x78
        [+0x080] User32Reserved   [Type: unsigned long [26]]
        [+0x0e8] UserReserved     [Type: unsigned long [5]]
        [+0x100] WOW32Reserved    : Unable to read memory at Address 0x100
        [+0x108] CurrentLocale    : Unable to read memory at Address 0x108
        [+0x10c] FpSoftwareStatusRegister : Unable to read memory at Address 0x10c
         ...
```

To display PEB associated with the current process use this command.

```
0: kd> dx -r2 @$curprocess.Environment
@$curprocess.Environment
    EnvironmentBlock [Type: _PEB]
        [+0x000] InheritedAddressSpace : Unable to read memory at Address 0x0
        [+0x001] ReadImageFileExecOptions : Unable to read memory at Address 0x1
        [+0x002] BeingDebugged    : Unable to read memory at Address 0x2
        [+0x003] BitField         : Unable to read memory at Address 0x3
        [+0x003 ( 0: 0)] ImageUsesLargePages : Unable to read memory at Address 0x3
        [+0x003 ( 1: 1)] IsProtectedProcess : Unable to read memory at Address 0x3
        [+0x003 ( 2: 2)] IsImageDynamicallyRelocated : Unable to read memory at Address 0x3
        [+0x003 ( 3: 3)] SkipPatchingUser32Forwarders : Unable to read memory at Address 0x3
        [+0x003 ( 4: 4)] IsPackagedProcess : Unable to read memory at Address 0x3
        [+0x003 ( 5: 5)] IsAppContainer   : Unable to read memory at Address 0x3
        [+0x003 ( 6: 6)] IsProtectedProcessLight : Unable to read memory at Address 0x3
        [+0x003 ( 7: 7)] IsLongPathAwareProcess : Unable to read memory at Address 0x3
        [+0x004] Padding0         [Type: unsigned char [4]]
        [+0x008] Mutant           : Unable to read memory at Address 0x8
        [+0x010] ImageBaseAddress : Unable to read memory at Address 0x10
        [+0x018] Ldr              : Unable to read memory at Address 0x18
        [+0x020] ProcessParameters : Unable to read memory at Address 0x20
        ...
```

Kernel Io.Handles object

Use the current process Io.Handles object to display kernel handle information.

```
0: kd> dx -r1 @$curprocess.Io.Handles
@$curprocess.Io.Handles
    [0x8]
    [0xc]
    [0x10]
    [0x14]
    [0x18]
    ...
```

Use the .First() function to display information about the first handle.

```
0: kd> dx -r2 @$curprocess.Io.Handles.First()
@$curprocess.Io.Handles.First()
    Handle           : 0x8
    Type             : Unexpected failure to dereference object
    GrantedAccess    : Unexpected failure to dereference object
    Object           [Type: _OBJECT_HEADER]
        [+0x000] PointerCount     : 228806 [Type: __int64]
        [+0x008] HandleCount      : 6 [Type: __int64]
        [+0x008] NextToFree       : 0x6 [Type: void *]
        [+0x010] Lock             [Type: _EX_PUSH_LOCK]
        [+0x018] TypeIndex        : 0xf2 [Type: unsigned char]
        [+0x019] TraceFlags       : 0x0 [Type: unsigned char]
        [+0x019 ( 0: 0)] DbgRefTrace      : 0x0 [Type: unsigned char]
        [+0x019 ( 1: 1)] DbgTracePermanent : 0x0 [Type: unsigned char]
        [+0x01a] InfoMask         : 0x0 [Type: unsigned char]
        [+0x01b] Flags            : 0x2 [Type: unsigned char]
        [+0x01b ( 0: 0)] NewObject        : 0x0 [Type: unsigned char]
        [+0x01b ( 1: 1)] KernelObject     : 0x1 [Type: unsigned char]
        [+0x01b ( 2: 2)] KernelOnlyAccess : 0x0 [Type: unsigned char]
        [+0x01b ( 3: 3)] ExclusiveObject  : 0x0 [Type: unsigned char]
        [+0x01b ( 4: 4)] PermanentObject  : 0x0 [Type: unsigned char]
        [+0x01b ( 5: 5)] DefaultSecurityQuota : 0x0 [Type: unsigned char]
        [+0x01b ( 6: 6)] SingleHandleEntry : 0x0 [Type: unsigned char]
        [+0x01b ( 7: 7)] DeletedInline    : 0x0 [Type: unsigned char]
        [+0x01c] Reserved         : 0x0 [Type: unsigned long]
        [+0x020] ObjectCreateInfo : 0xfffff801f6d9c6c0 [Type: _OBJECT_CREATE_INFORMATION *]
        [+0x020] QuotaBlockCharged : 0xfffff801f6d9c6c0 [Type: void *]
        [+0x028] SecurityDescriptor : 0xffffb984aa815d06 [Type: void *]
        [+0x030] Body             [Type: _QUAD]
        ObjectType       : Unexpected failure to dereference object
        UnderlyingObject : Unexpected failure to dereference object
```

Note that the Io.Handles object is a kernel only object.

Working around symbol file limitations with casting

When displaying information about various Windows system variables, there are times where not all of the type information is available in the public symbols. This example illustrates this situation.

```
0: kd> dx nt!PsIdleProcess
Error: No type (or void) for object at Address 0xfffff800e1d50128
```

The dx command supports the ability to reference the address of a variable which does not have type information. Such "address of" references are treated as "void *" and can be cast as such. This means that if the data type is known, the following syntax can be used to display type information for the variable.

```
dx (Datatype *)&VariableName
```

For example for a nt!PsIdleProcess which has a data type of nt!_EPROCESS, use this command.

```
dx (nt!_EPROCESS *)&nt!PsIdleProcess
(nt!_EPROCESS *)&nt!PsIdleProcess                : 0xfffff800e1d50128 [Type: _EPROCESS *]
    [+0x000] Pcb              [Type: _KPROCESS]
    [+0x2c8] ProcessLock      [Type: _EX_PUSH_LOCK]
    [+0x2d0] CreateTime       : {4160749568} [Type: _LARGE_INTEGER]
    [+0x2d8] RundownProtect   [Type: _EX_RUNDOWN_REF]
    [+0x2e0] UniqueProcessId  : 0x1000 [Type: void *]
    [+0x2e8] ActiveProcessLinks [Type: _LIST_ENTRY]
    [+0x2f8] Flags2           : 0x218230 [Type: unsigned long]
    [+0x2f8 ( 0: 0)] JobNotReallyActive : 0x0 [Type: unsigned long]
```

The dx command does not support switching expression evaluators with the @@ MASM syntax. For more information about expression evaluators, see Evaluating Expressions.

Using LINQ With the debugger objects

LINQ syntax can be used with the debugger objects to search and manipulate data. LINQ is conceptually similar to the Structured Query Language (SQL) that is used to query databases. You can use a number of LINQ methods to search, filter and parse debug data. For information us using LINQ with the debugger objects, see Using LINQ With the debugger objects.

Using debugger objects with NatVis and JavaScript

For information about using debugger objects with NatVis, see Native Debugger Objects in NatVis.

For information about using debugger objects with JavaScript, see Native Debugger Objects in JavaScript Extensions.

See also

Using LINQ With the debugger objects

Native Debugger Objects in NatVis

Native Debugger Objects in JavaScript Extensions

e, ea, eb, ed, eD, ef, ep, eq, eu, ew, eza (Enter Values)

3/5/2019 • 2 minutes to read • Edit Online

The **e*** commands enter into memory the values that you specify.

This command should not be confused with the **~E (Thread-Specific Command)** qualifier.

```
e{b|d|D|f|p|q|w} Address [Values]
e{a|u|za|zu} Address "String"
e Address [Values]
```

Parameters

Syntax eD ef

Address
Specifies the starting address where to enter values. The debugger replaces the value at *Address* and each subsequent memory location until all *Values* have been used.

Values
Specifies one or more values to enter into memory. Multiple numeric values should be separated with spaces. If you do not specify any values, the current address and the value at that address will be displayed, and you will be prompted for input.

String
Specifies a string to be entered into memory. The **ea** and **eza** commands will write this to memory as an ASCII string; the **eu** and **ezu** commands will write this to memory as a Unicode string. The **eza** and **ezu** commands write a terminal **NULL**; the **ea** and **eu** commands do not. *String* must be enclosed in quotation marks.

Environment

Modes	user mode, kernel mode
Targets	live, crash dump
Platforms	all

Additional Information

For an overview of memory manipulation and a description of other memory-related commands, see Reading and Writing Memory.

Remarks

This command exists in the following forms. The second characters of the **ed** and **eD** commands are case-sensitive.

COMMAND	ENTER
e	This enters data in the same format as the most recent **e** command. (If the most recent **e** command was **ea**, **eza**, **eu**, or **ezu**, the final parameter will be *String* and may not be omitted.)
ea	ASCII string (not NULL-terminated).
eb	Byte values.
ed	Double-word values (4 bytes).
eD	Double-precision floating-point numbers (8 bytes).
ef	Single-precision floating-point numbers (4 bytes).
ep	Pointer-sized values. This command is equivalent to **ed** or **eq**, depending on whether the target computer's processor architecture is 32-bit or 64-bit, respectively.
eq	Quad-word values (8 bytes).
eu	Unicode string (not NULL-terminated).
ew	Word values (2 bytes).
eza	NULL-terminated ASCII string.
ezu	NULL-terminated Unicode string.

Numeric values will be interpreted as numbers in the current radix (16, 10, or 8). To change the default radix, use the **n (Set Number Base)** command. The default radix can be overridden by specifying the **0x** prefix (hexadecimal), the **0n** prefix (decimal), the **0t** prefix (octal), or the **0y** prefix (binary).

Note The default radix behaves differently when C++ expressions are being used. See Evaluating Expressions for details.

When entering byte values with the **eb** command, you can use single straight quotation marks to specify characters. If you want to include multiple characters, each must be surrounded with its own quotation marks. This allows you to enter a character string that is not terminated by a null character. For example:

```
eb 'h' 'e' 'l' 'l' 'o'
```

C-style escape characters (such as '\0' or '\n') may not be used with these commands.

If you omit the *Values* parameter, you will be prompted for input. The address and its current contents will be displayed, and an **Input>** prompt will appear. You can then do any of the following:

- Enter a new value, by typing the value and pressing ENTER.
- Preserve the current value in memory by pressing SPACE followed by ENTER.
- Exit from the command by pressing ENTER.

f, fp (Fill Memory)

12/5/2018 • 2 minutes to read • Edit Online

The **f** and **fp** commands fill the specified memory range with a repeating pattern.

These commands should not be confused with the **~F (Freeze Thread)** command.

```
f Range Pattern
fp [MemoryType] PhysicalRange Pattern
```

Parameters

Range
Specifies the range in virtual memory to fill. For more syntax details, see Address and Address Range Syntax.

PhysicalRange
(Kernel mode only) Specifies the range in physical memory to fill. The syntax of *PhysicalRange* is the same as that of a virtual memory range, except that no symbol names are permitted.

MemoryType
(Kernel mode only) Specifies the type of physical memory, which can be one of the following:

[c]
Cached memory.

[uc]
Uncached memory.

[wc]
Write-combined memory.

Pattern
Specifies one or more byte values with which to fill memory.

Environment

Modes	**f:** user mode, kernel mode **fp:** kernel mode only
Targets	live, crash dump
Platforms	all

Additional Information

For an overview of memory manipulation and a description of other memory-related commands, see Reading and Writing Memory.

Remarks

This command fills the memory area specified by *range* with the specified *pattern*, repeated as many times as necessary.

The *pattern* parameter must be input as a series of bytes. These can be entered as numeric or as ASCII characters.

Numeric values will be interpreted as numbers in the current radix (16, 10, or 8). To change the default radix, use the **n (Set Number Base)** command. The default radix can be overridden by specifying the **0x** prefix (hexadecimal), the **0n** prefix (decimal), the **0t** prefix (octal), or the **0y** prefix (binary).

Note The default radix behaves differently when C++ expressions are being used. For more information, see the Evaluating Expressions topic.

If ASCII characters are used, each character must be enclosed in single straight quotation marks. C-style escape characters (such as '\0' or '\n') may not be used.

If multiple bytes are specified, they must be separated by spaces.

If *pattern* has more values than the number of bytes in the range, the debugger ignores the extra values.

Here are some examples. Assuming the current radix is 16, the following command will fill memory locations 0012FF40 through 0012FF5F with the pattern "ABC", repeated several times:

```
0:000> f 0012ff40 L20 'A' 'B' 'C'
```

The following command has the exact same effect:

```
0:000> f 0012ff40 L20 41 42 43
```

The following examples show how you can use the physical memory types (**c**, **uc**, and **wc**) with the **fp** command in kernel mode:

```
kd> fp [c] 0012ff40 L20 'A' 'B' 'C'
```

```
kd> fp [uc] 0012ff40 L20 'A' 'B' 'C'
```

```
kd> fp [wc] 0012ff40 L20 'A' 'B' 'C'
```

g (Go)

12/5/2018 • 2 minutes to read • Edit Online

The **g** command starts executing the given process or thread. Execution will halt at the end of the program, when *BreakAddress* is hit, or when another event causes the debugger to stop.

User-Mode Syntax

```
[~Thread] g[a] [= StartAddress] [BreakAddress ... [; BreakCommands]]
```

Kernel-Mode Syntax

```
g[a] [= StartAddress] [BreakAddress ... [; BreakCommands]]
```

Parameters

Thread
(User mode only) Specifies the thread to execute. For syntax details, see Thread Syntax.

a
Causes any breakpoint created by this command to be a processor breakpoint (like those created by **ba**) rather than a software breakpoint (like those created by **bp** and **bm**). If *BreakAddress* is not specified, no breakpoint is created and the **a** flag has no effect.

StartAddress
Specifies the address where execution should begin. If this is not specified, the debugger passes execution to the address specified by the current value of the instruction pointer. For more syntax details, see Address and Address Range Syntax.

BreakAddress
Specifies the address for a breakpoint. If *BreakAddress* is specified, it must specify an instruction address (that is, the address must contain the first byte of an instruction). Up to ten break addresses, in any order, can be specified at one time. If *BreakAddress* cannot be resolved, it is stored as an unresolved breakpoint. For more syntax details, see Address and Address Range Syntax.

BreakCommands
Specifies one or more commands to be automatically executed when the breakpoint specified by *BreakAddress* is hit. The *BreakCommands* parameter must be preceded by a semicolon. If multiple *BreakAddress* values are specified, *BreakCommands* applies to all of them.

Note The *BreakCommands* parameter is only available when you are embedding this command within a command string used by another command -- for example, within another breakpoint command or within an except or event setting. On a command line, the semicolon will terminate the **g** command, and any additional commands listed after the semicolon will be executed immediately after the **g** command is done.

Environment

Modes	user mode, kernel mode

Targets	live debugging only
Platforms	all

Additional Information

For other methods of issuing this command and an overview of related commands, see Controlling the Target.

Remarks

If *Thread* is specified, then the **g** command is executed with the specified thread unfrozen and all others frozen. For example, if the **~123g**, **~#g**, or **~*g** command is specified, the specified threads are unfrozen and all others are frozen.

gc (Go from Conditional Breakpoint)

12/5/2018 • 2 minutes to read • Edit Online

The **gc** command resumes execution from a conditional breakpoint in the same fashion that was used to hit the breakpoint (stepping, tracing, or freely executing).

```
gc
```

Environment

Modes	user mode, kernel mode
Targets	live debugging only
Platforms	all

Additional Information

For an overview of related commands, see Controlling the Target.

Remarks

When a conditional breakpoint includes an execution command at the end, this should be the **gc** command.

For example, the following is a proper conditional breakpoint formulation:

```
0:000> bp Address "j (Condition) 'OptionalCommands'; 'gc' "
```

When this breakpoint is encountered and the expression is false, execution will resume using the same execution type that was previously used. For example, if you used a **g (Go)** command to reach this breakpoint, execution would resume freely. But if you reached this breakpoint while stepping or tracing, execution would resume with a step or a trace.

On the other hand, the following is an improper breakpoint formulation, since execution will always resume freely even if you had been stepping before reaching the breakpoint:

```
0:000> bp Address "j (Condition) 'OptionalCommands'; 'g' "
```

gh (Go with Exception Handled)

12/5/2018 • 2 minutes to read • Edit Online

The **gh** command marks the given thread's exception as having been handled and allows the thread to restart execution at the instruction that caused the exception.

User-Mode Syntax

```
[~Thread] gh[a] [= StartAddress] [BreakAddress ... [; BreakCommands]]
```

Kernel-Mode Syntax

```
gh[a] [= StartAddress] [BreakAddress ... [; BreakCommands]]
```

Parameters

Thread
(User mode only) Specifies the thread to execute. This thread must have been stopped by an exception. For syntax details, see Thread Syntax.

a
Causes any breakpoint created by this command to be a processor breakpoint (like those created by **ba**) rather than a software breakpoint (like those created by **bp** and **bm**). If *BreakAddress* is not specified, no breakpoint is created and the **a** flag has no effect.

StartAddress
Specifies the address at which execution should begin. If this is not specified, the debugger passes execution to the address where the exception occurred. For more syntax details, see Address and Address Range Syntax.

BreakAddress
Specifies the address for a breakpoint. If *BreakAddress* is specified, it must specify an instruction address (that is, the address must contain the first byte of an instruction). Up to ten break addresses, in any order, can be specified at one time. If *BreakAddress* cannot be resolved, it is stored as an unresolved breakpoint. For more syntax details, see Address and Address Range Syntax.

BreakCommands
Specifies one or more commands to be automatically executed when the breakpoint specified by *BreakAddress* is hit. The *BreakCommands* parameter must be preceded by a semicolon. If multiple *BreakAddress* values are specified, *BreakCommands* applies to all of them.

Note The *BreakCommands* parameter is only available when you are embedding this command within a command string used by another command -- for example, within another breakpoint command or within an except or event setting. On a command line, the semicolon will terminate the **gh** command, and any additional commands listed after the semicolon will be executed immediately after the **gh** command is done.

Environment

Modes	user mode, kernel mode

Targets	live debugging only
Platforms	all

Additional Information

For other methods of issuing this command and an overview of related commands, see Controlling the Target.

Remarks

If you use the *BreakAddress* parameter to set a breakpoint, this new breakpoint will only be triggered by the current thread. Other threads that execute the code at that location will not be stopped.

If *Thread* is specified, then the **gh** command is executed with the specified thread unfrozen and all others frozen. For example, if the **~123gh**, **~#gh**, or **~*gh** command is specified, the specified threads are unfrozen and all others are frozen.

gn, gN (Go with Exception Not Handled)

12/5/2018 • 2 minutes to read • Edit Online

The **gn** and **gN** commands continue execution of the given thread without marking the exception as having been handled. This allows the application's exception handler to handle the exception.

User-Mode Syntax

```
[~Thread] gn[a] [= StartAddress] [BreakAddress ... [; BreakCommands]]
[~Thread] gN[a] [= StartAddress] [BreakAddress ... [; BreakCommands]]
```

Kernel-Mode Syntax

```
gn[a] [= StartAddress] [BreakAddress ... [; BreakCommands]]
gN[a] [= StartAddress] [BreakAddress ... [; BreakCommands]]
```

Parameters

Thread
(User mode only) Specifies the thread to execute. This thread must have been stopped by an exception. For syntax details, see Thread Syntax.

a
Causes any breakpoint created by this command to be a processor breakpoint (like those created by **ba**) rather than a software breakpoint (like those created by **bp** and **bm**). If *BreakAddress* is not specified, no breakpoint is created and the **a** flag has no effect.

StartAddress
Specifies the address where execution should begin. If this is not specified, the debugger passes execution to the address where the exception occurred. For more syntax details, see Address and Address Range Syntax.

BreakAddress
Specifies the address for a breakpoint. If *BreakAddress* is specified, it must specify an instruction address (that is, the address must contain the first byte of an instruction). Up to ten break addresses, in any order, can be specified at one time. If *BreakAddress* cannot be resolved, it is stored as an unresolved breakpoint. For more syntax details, see Address and Address Range Syntax.

BreakCommands
Specifies one or more commands to be automatically executed when the breakpoint specified by *BreakAddress* is hit. The *BreakCommands* parameter must be preceded by a semicolon. If multiple *BreakAddress* values are specified, *BreakCommands* applies to all of them.

Note The *BreakCommands* parameter is only available when you are embedding this command within a command string used by another command -- for example, within another breakpoint command or within an except or event setting. On a command line, the semicolon will terminate the command, and any additional commands listed after the semicolon will be executed immediately after the **gn** or **gN** command is done.

Environment

Modes	user mode, kernel mode

Targets	live debugging only
Platforms	all

Additional Information

For other methods of issuing this command and an overview of related commands, see Controlling the Target.

Remarks

If the debugger is not stopped at a breakpoint, **gn** and **gN** behave identically. If the debugger is stopped at a breakpoint, **gn** will not work; you must capitalize the "N" to execute this command. This is a safety precaution, since it is rarely wise to continue a breakpoint unhandled.

If you use the *BreakAddress* parameter to set a breakpoint, this new breakpoint will only be triggered by the current thread. Other threads that execute the code at that location will not be stopped.

If *Thread* is specified, then the **gn** command is executed with the specified thread unfrozen and all others frozen. For example, if the **~123gn**, **~#gn**, or **~*gn** command is specified, the specified threads are unfrozen and all others are frozen.

gu (Go Up)

12/5/2018 • 2 minutes to read • Edit Online

The **gu** command causes the target to execute until the current function is complete.

User-Mode Syntax

```
[~Thread] gu
```

Kernel-Mode Syntax

```
gu
```

Parameters

Thread
(User mode only) Specifies the thread to execute. This thread must have been stopped by an exception. For syntax details, see Thread Syntax.

Environment

Modes	user mode, kernel mode
Targets	live debugging only
Platforms	all

Additional Information

For other methods of issuing this command and an overview of related commands, see Controlling the Target.

Remarks

The **gu** command executes the target until the current function call returns.

If the current function is called recursively, the **gu** command will not halt execution until the *current instance* of the current function returns. In this way, **gu** differs from **g @$ra**, which will halt any time the return address of this function is hit.

Note The **gu** command distinguishes different instances of a function by measuring the call stack depth. Executing this command in assembly mode after the arguments have been pushed to the stack and just before the call is made may cause this measurement to be incorrect. Function returns that are optimized away by the compiler may similarly cause this command to stop at the wrong instance of this return. These errors are rare, and can only happen during recursive function calls.

If *Thread* is specified, then the **gu** command is executed with the specified thread unfrozen and all others frozen. For example, if the **~123gu**, **~#gu**, or **~*gu** command is specified, the specified threads are unfrozen and all others are frozen.

ib, iw, id (Input from Port)

12/5/2018 • 2 minutes to read • Edit Online

The **ib**, **iw**, and **id** commands read and display a byte, word, or double word from the selected port.

```
ib Address
iw Address
id Address
```

Parameters

Address
The address of the port.

Environment

Modes	Kernel mode only
Targets	Live debugging only
Platforms	x86-based computer only

Remarks

The **ib** command reads a single byte, the **iw** command reads a word, and the **id** command reads a double word.

Make sure that reading an I/O port does not affect the behavior of the device that you are reading from. Some devices change state after a read-only port has been read. You should also not try to read a word or double-word from a port that does not allow values of this length.

See also

ob, od, ow (Output to Port)

j (Execute If - Else)

12/5/2018 • 2 minutes to read • Edit Online

The **j** command conditionally executes one of the specified commands, depending on the evaluation of a given expression.

```
j Expression Command1 ; Command2
j Expression 'Command1' ; 'Command2'
```

Parameters

Expression
The expression to evaluate. If this expression evaluates to a nonzero value, *Command1* is executed. If this expression evaluates to zero, *Command2* is executed. For more information about the syntax of this expression, see Numerical Expression Syntax.

Command1
The command string to be executed if the expression in *Expression* evaluates to a nonzero value (TRUE). You can combine multiple commands by surrounding the command string with single straight quotation marks (') and separating commands by using semicolons. If the command string is a single command, the single quotation marks are optional.

Command2
The command string to be executed if the expression in *Expression* evaluates to zero (FALSE). You can combine multiple commands by surrounding the command string with single straight quotation marks (') and separating commands by using semicolons. If the command string is a single command, the single quotation marks are optional.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

You cannot add a semicolon or additional commands after the **j** command. If a semicolon appears after *Command2*, everything after the semicolon is ignored.

The following command displays the value of **eax** if **MySymbol** is equal to zero and displays the values of **ebx** and **ecx** otherwise.

```
0:000> j (MySymbol=0) 'r eax'; 'r ebx; r ecx'
```

You could omit the single quotation marks around **r eax**, but they make the command easier to read. If you want

to omit one of the commands, you can include empty quotation marks or omit the parameter for that command, as in the following commands.

```
0:000> j (MySymbol=0) ''; 'r ebx; r ecx'
0:000> j (MySymbol=0)  ; 'r ebx; r ecx'
```

You can also use the **j** command inside other commands. For example, you can use a **j** command to create conditional breakpoints.

```
0:000> bp `mysource.cpp:143` "j (poi(MyVar)>0n20) ''; 'gc' "
```

For more information about the syntax for conditional breakpoints, see Setting a Conditional Breakpoint.

See also

z (Execute While)

k, kb, kc, kd, kp, kP, kv (Display Stack Backtrace)

7/9/2019 • 5 minutes to read • Edit Online

The **k*** commands display the stack frame of the given thread, together with related information..

User-Mode, x86 Processor

```
[~Thread] k[b|p|P|v] [c] [n] [f] [L] [M] [FrameCount]
[~Thread] k[b|p|P|v] [c] [n] [f] [L] [M] = BasePtr [FrameCount]
[~Thread] k[b|p|P|v] [c] [n] [f] [L] [M] = BasePtr StackPtr InstructionPtr
[~Thread] kd [WordCount]
```

Kernel-Mode, x86 Processor

```
[Processor] k[b|p|P|v] [c] [n] [f] [L] [M] [FrameCount]
[Processor] k[b|p|P|v] [c] [n] [f] [L] [M] = StackPtr FrameCount
[Processor] k[b|p|P|v] [c] [n] [f] [L] [M] = BasePtr StackPtr InstructionPtr
[Processor] kd [WordCount]
```

User-Mode, x64 Processor

```
[~Thread] k[b|p|P|v] [c] [n] [f] [L] [M] [FrameCount]
[~Thread] k[b|p|P|v] [c] [n] [f] [L] [M] = StackPtr FrameCount
[~Thread] k[b|p|P|v] [c] [n] [f] [L] [M] = StackPtr InstructionPtr FrameCount
[~Thread] kd [WordCount]
```

Kernel-Mode, x64 Processor

```
[Processor] k[b|p|P|v] [c] [n] [f] [L] [M] [FrameCount]
[Processor] k[b|p|P|v] [c] [n] [f] [L] [M] = StackPtr FrameCount
[Processor] k[b|p|P|v] [c] [n] [f] [L] [M] = StackPtr InstructionPtr FrameCount
[Processor] kd [WordCount]
```

User-Mode, ARM Processor

```
[~Thread] k[b|p|P|v] [c] [n] [f] [L] [M] [FrameCount]
[~Thread] k[b|p|P|v] [c] [n] [f] [L] [M] = StackPtr FrameCount
[~Thread] k[b|p|P|v] [c] [n] [f] [L] [M] = StackPtr InstructionPtr FrameCount
[~Thread] kd [WordCount]
```

Kernel-Mode, ARM Processor

```
[Processor] k[b|p|P|v] [c] [n] [f] [L] [M] [FrameCount]
[Processor] k[b|p|P|v] [c] [n] [f] [L] [M] = StackPtr FrameCount
[Processor] k[b|p|P|v] [c] [n] [f] [L] [M] = StackPtr InstructionPtr FrameCount
[Processor] kd [WordCount]
```

Parameters

Thread

Specifies the thread whose stack is to be displayed. If you omit this parameter, the stack of the current thread is displayed. For more information about thread syntax, see Thread Syntax. You can specify threads only in user mode.

Processor
Specifies the processor whose stack is to be displayed. For more information about processor syntax, see Multiprocessor Syntax.

b
Displays the first three parameters that are passed to each function in the stack trace.

c
Displays a clean stack trace. Each display line includes only the module name and the function name.

p
Displays all of the parameters for each function that is called in the stack trace. The parameter list includes each parameter's data type, name, and value. The p option is case sensitive. This parameter requires full symbol information.

P
Displays all of the parameters for each function that is called in the stack trace, like the p parameter. However, for P, the function parameters are printed on a second line of the display, instead of on the same line as the rest of the data.

v
Displays frame pointer omission (FPO) information. On x86-based processors, the display also includes calling convention information.

n
Displays frame numbers.

f
Displays the distance between adjacent frames. This distance is the number of bytes that separate the frames on the actual stack.

L
Hides source lines in the display. L is case sensitive.

M
Displays output using Debugger Markup Language. Each frame number in the display is a link that you can click to set the local context and display local variables. For information about the local context, see **.frame**.

FrameCount
Specifies the number of stack frames to display. You should specify this number in hexadecimal format, unless you have changed the radix by using the **n (Set Number Base)** command. The default value is 20 (0x14), unless you have changed the default value by using the **.kframes (Set Stack Length)** command.

BasePtr
Specifies the base pointer for the stack trace. The *BasePtr* parameter is available only if there is an equal sign (=) after the command.

StackPtr
Specifies the stack pointer for the stack trace. If you omit *StackPtr* and *InstructionPtr*, the command uses the stack pointer that the rsp (or esp) register specifies and the instruction pointer that the rip (or eip) register specifies.

InstructionPtr
Specifies the instruction pointer for the stack trace. If you omit *StackPtr* and *InstructionPtr*, the command uses the stack pointer that the rsp (or esp) register specifies and the instruction pointer that the rip (or eip) register specifies.

WordCount
Specifies the number of DWORD_PTR values in the stack to dump. The default value is 20 (0x14), unless you changed the default value by using the **.kframes (Set Stack Length)** command.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about the register context and other context settings, see Changing Contexts.

Remarks

When you issue the **k**, **kb**, **kp**, **kP**, or **kv** command, a stack trace is displayed in a tabular format. If line loading is enabled, source modules and line numbers are also displayed.

The stack trace includes the base pointer for the stack frame, the return address, and function names.

If you use the **kp** or **kP** command, the full parameters for each function that is called in the stack trace are displayed. The parameter list includes each parameter's data type, name, and value.

This command might be slow. For example, when **MyFunction1** calls **MyFunction2**, the debugger must have full symbol information for **MyFunction1** to display the parameters that are passed in this call. This command does not fully display internal Microsoft Windows routines that are not exposed in public symbols.

If you use the **kb** or **kv** command, the first three parameters that are passed to each function are displayed. If you use the **kv** command, FPO data is also displayed.

On an x86-based processor, the **kv** command also displays calling convention information.

When you use the **kv** command, the FPO information is added at the end of the line in the following format.

FPO TEXT	MEANING
FPO: [non-Fpo]	No FPO data for the frame.
FPO: [N1,N2,N3]	*N1* is the total number of parameters. *N2* is the number of DWORD values for the local variables. *N3* is the number of registers that are saved.

FPO TEXT	MEANING
FPO: [N1,N2] TrapFrame @ Address	*N1* is the total number of parameters. *N2* is the number of DWORD values for the locals. *Address* is the address of the trap frame.
FPO: TaskGate Segment:0	*Segment* is the segment selector for the task gate.
FPO: [EBP 0xBase]	*Base* is the base pointer for the frame.

The **kd** command displays the raw stack data. Each DWORD value is displayed on a separate line. Symbol information is displayed for those lines together with associated symbols. This format creates a more detailed list than the other **k*** commands. The **kd** command is equivalent to a **dds (Display Memory)** command that uses the stack address as its parameter.

If you use the **k** command at the beginning of a function (before the function prolog has been executed), you receive incorrect results. The debugger uses the frame register to compute the current backtrace, and this register is not set correctly for a function until its prolog has been executed.

In user mode, the stack trace is based on the stack of the current thread. For more information about threads, see Controlling Processes and Threads.

In kernel mode, the stack trace is based on the current register context. You can set the register context to match a specific thread, context record, or trap frame.

Additional Information

For more information about the register context and other context settings, see Changing Contexts.

l+, l- (Set Source Options)

10/8/2019 • 2 minutes to read • Edit Online

The **l+**and **l-** commands set the source line options that control source display and program stepping options.

```
l+Option
l-Option
l{+|-}
```

Parameters

+ or **-**
Specifies whether a given option is to be turned on (plus sign [+])or turned off (minus sign [-]).

Option One of the following options. The options must be in lowercase letters.

l
Displays source line numbers at the command prompt. You can disable source line display through l-ls or .prompt_allow -src. To make the source line numbers visible, you must enable source line display through both mechanisms.

o
Hides all messages (other than the source line and line number) when you are stepping through code. (The **s** option must also be active for the **o** option to have any effect.)

s
Displays source lines and source line numbers at the command prompt.

t
Starts source mode. If this mode is not set, the debugger is in assembly mode.

*****
Turns on or turns off all options.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about source debugging and related commands, see Debugging in Source Mode. For more information about assembly debugging and related commands, see Debugging in Assembly Mode.

Remarks

If you omit *Option*, the previously set options are displayed. In this case, the **l+** and **l-** commands have identical effects. However, you must include a plus sign (**+**) or minus sign (-) for the **l** command to work.

You can include only one *Option* every time that you issue this command. If you list more than one option, only the first option is detected. However, by repeatedly issuing this command, you can turn on or off as many options as you want. (In other words, **l+lst** does not work, but **l+l; l+s; l+t** does achieve the effect that you want.)

When you specify the **s** option, source lines and line numbers are displayed when you step through code, regardless of whether you specified the **l** option. The **o** option has no effect unless you specify the **s** option.

Source line options do not take effect unless you enable line number loading by using the **.lines (Toggle Source Line Support)** command or the -lines command-line option. By default, if you have not used these commands, WinDbg turns on source line support and CDB turns it off.

ld (Load Symbols)

12/5/2018 • 2 minutes to read • Edit Online

The **ld** command loads symbols for the specified module and updates all module information.

```
ld ModuleName [/f FileName]
```

Parameters

ModuleName
Specifies the name of the module whose symbols are to be loaded. *ModuleName* can contain a variety of wildcard characters and specifiers.

/f *FileName*
Changes the name selected for the match. By default the module name is matched, but when **/f** is used the file name is matched instead of the module name. *FileName* can contain a variety of wildcard characters and specifiers. For more information on the syntax of wildcard characters and specifiers, see String Wildcard Syntax.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

The debugger's default behavior is to use *lazy symbol loading* (also known as deferred symbol loading). This means that symbols are not actually loaded until they are needed.

The **ld** command, on the other hand, forces all symbols for the specified module to be loaded.

Additional Information

For more information about deferred (lazy) symbol loading, see Deferred Symbol Loading. For more information about other symbol options, see Setting Symbol Options.

lm (List Loaded Modules)

12/5/2018 • 4 minutes to read • Edit Online

The **lm** command displays the specified loaded modules. The output includes the status and the path of the module.

```
lm Options [a Address] [m Pattern | M Pattern]
```

Parameters

Options
Any combination of the following options:

D
Displays output using Debugger Markup Language.

o
Displays only loaded modules.

l
Displays only modules whose symbol information has been loaded.

v
Causes the display to be verbose. The display includes the symbol file name, the image file name, checksum information, version information, date stamps, time stamps, and information about whether the module is managed code (CLR). This information is not displayed if the relevant headers are missing or paged out.

u
(Kernel mode only) Displays only user-mode symbol information.

k
(Kernel mode only) Displays only kernel-mode symbol information.

e
Displays only modules that have a symbol problem. These symbols include modules that have no symbols and modules whose symbol status is C, T, #, M, or Export. For more information about these notations, see Symbol Status Abbreviations.

c
Displays checksum data.

1m
Reduces the output so that nothing is included except the names of the modules. This option is useful if you are using the **.foreach** token to pipe the command output into another command's input.

sm
Sorts the display by module name instead of by the start address.

In addition, you can include only one of the following options. If you do not include any of these options, the display includes the symbol file name.

i
Displays the image file name.

f
Displays the full image path. (This path always matches the path that is displayed in the initial load notification, unless you issued a **.reload -s** command.) When you use f, symbol type information is not displayed.

n
Displays the image name. When you use n, symbol type information is not displayed.

p
Displays the mapped image name. When you use p, symbol type information is not displayed.

t
Displays the file time stamps. When you use t, symbol type information is not displayed.

a *Address*
Specifies an address that is contained in this module. Only the module that contains this address is displayed. If Address contains an expression, it must be enclosed in parentheses.

m *Pattern*
Specifies a pattern that the module name must match. Pattern can contain a variety of wildcard characters and specifiers. For more information about the syntax of this information, see String Wildcard Syntax.

Note In most cases, the module name is the file name without the file name extension. For example, if you want to display information about the Flpydisk.sys driver, use the lm mflpydisk command, not lm mflpydisk.sys. In some cases, the module name differs significantly from the file name.

M *Pattern*
Specifies a pattern that the image path must match. Pattern can contain a variety of wildcard characters and specifiers. For more information about the syntax of this information, see String Wildcard Syntax.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

The **lm** command lists all of the modules and the status of symbols for each module.

Microsoft Windows Server 2003 and later versions of Windows maintain an unloaded module list for user-mode processes. When you are debugging a user-mode process or dump file, the **lm** command also shows these unloaded modules.

This command shows several columns or fields, each with a different title. Some of these titles have specific meanings:

- *module name* is typically the file name without the file name extension. In some cases, the module name differs significantly from the file name.
- The symbol type immediately follows the module name. This column is not labeled. For more information about the various status values, see Symbol Status Abbreviations. If you have loaded symbols, the symbol file name follows this column.

- The first address in the module is shown as start. The first address after the end of the module is shown as end. For example, if start is "faab4000" and end is "faab8000", the module extends from 0xFAAB4000 to 0xFAAB7FFF, inclusive.
- **lmv** only: The image path column shows the name of the executable file, including the file name extension. Typically, the full path is included in user mode but not in kernel mode.
- **lmv** only: The loaded symbol image file value is the same as the image name, unless Microsoft CodeView symbols are present.
- **lmv** only: The mapped memory image file value is typically not used. If the debugger is mapping an image file (for example, during minidump debugging), this value is the name of the mapped image.

The following code example shows the **lm** command with a Windows Server 2003 target computer. This example includes the m and s* options, so only modules that begin with "s" are displayed.

```
kd> lm m s*
start    end        module name
f9f73000 f9f7fd80   sysaudio     (deferred)
fa04b000 fa09b400   srv          (deferred)
faab7000 faac8500   sr           (deferred)
facac000 facbae00   serial       (deferred)
fb008000 fb00ba80   serenum      e:\mysymbols\SereEnum.pdb\.......
fb24f000 fb250000   swenum       (deferred)

Unloaded modules:
f9f53000 f9f61000   swmidi.sys
fb0ae000 fb0b0000   splitter.sys
fb040000 fb043000   Sfloppy.SYS
```

Examples

The following two examples show the **lm** command once without any options and once with the sm option. Compare the sort order in the two examples.

Example 1:

```
0:000> lm
start    end        module name
01000000 0100d000   stst       (deferred)
77c10000 77c68000   msvcrt     (deferred)
77dd0000 77e6b000   ADVAPI32   (deferred)
77e70000 77f01000   RPCRT4     (deferred)
7c800000 7c8f4000   kernel32   (deferred)
7c900000 7c9b0000   ntdll      (private pdb symbols) c:\db20sym\ntdll.pdb
```

Example 2:

```
0:000> lmsm
start    end        module name
77dd0000 77e6b000   ADVAPI32   (deferred)
7c800000 7c8f4000   kernel32   (deferred)
77c10000 77c68000   msvcrt     (deferred)
7c900000 7c9b0000   ntdll      (private pdb symbols)  c:\db20sym\ntdll.pdb
77e70000 77f01000   RPCRT4     (deferred)
01000000 0100d000   stst       (deferred)
```

ln (List Nearest Symbols)

12/5/2018 • 2 minutes to read • Edit Online

The **ln** command displays the symbols at or near the given address.

```
ln Address
ln /D Address
```

Parameters

Address
Specifies the address where the debugger should start to search for symbols. The nearest symbols, either before or after *Address*, are displayed. For more information about the syntax, see Address and Address Range Syntax.

/D
Specifies that the output is displayed using Debugger Markup Language (DML). The DML output includes a link that you can use to explore the module that contains the nearest symbol. It also includes a link that you can use to set a breakpoint.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

You can use the **ln** command to help determine what a pointer is pointing to. This command can also be useful when you are looking at a corrupted stack to determine which procedure made a call.

If source line information is available, the **ln** display also includes the source file name and line number information.

If you are using a source server, the **ln** command displays information that is related to the source server.

ls, lsa (List Source Lines)

12/5/2018 • 2 minutes to read • Edit Online

The **ls** and **lsa** commands display a series of lines from the current source file and advance the current source line number.

```
ls [.] [first] [, count]
lsa [.] address [, first [, count]]
```

Parameters

.
Causes the command to look for the source file that the debugger engine or the **.srcpath (Set Source Path)** command are using. If the period (**.**) is not included, **ls** uses the file that was most recently loaded with the **lsf (Load Source File)** command.

address
Specifies the address where the source display is to begin.

For more information about the syntax, see Address and Address Range Syntax.

first
Specifies the first line to display. The default value is the current line.

count
Specifies the quantity of lines to display. The default value is 20 (0x14), unless you have changed the default value by using the **lsp -a** command.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

After you run the **ls** or **lsa** command, the current line is redefined as the final line that is displayed plus one. The current line is used in future **ls**, **lsa**, and **lsc** commands.

See also

lsc (List Current Source)

lsf, lsf- (Load or Unload Source File)

lsc (List Current Source)

12/5/2018 • 2 minutes to read • Edit Online

The **lsc** command displays the current source file name and line number.

```
lsc
```

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

See also

ls, lsa (List Source Lines)

lsf, lsf- (Load or Unload Source File)

lse (Launch Source Editor)

12/5/2018 • 2 minutes to read • Edit Online

The **lse** command opens an editor for the current source file.

```
lse
```

Environment

Modes	User-mode, kernel-mode
Targets	Live, crash dump
Platforms	All

Remarks

The **lse** command opens an editor for the current source file. This command is equivalent to clicking **Edit this file** in the shortcut menu of the Source window in WinDbg.

The editor is opened on the computer that the target is running on, so you cannot use the **lse** command from a remote client.

The WinDiff editor registry information or the value of the WINDBG_INVOKE_EDITOR environment variable determine which editor is opened. For example, consider the following value of WINDBG_INVOKE_EDITOR.

```
c:\my\path\myeditor.exe -file %f -line %l
```

This value indicates that Myeditor.exe opens to the one-based line number of the current source file. The **%l** option indicates that line numbers should be read as one-based, and **%f** indicates that the current source file should be used. You could also include **%L** to indicate that line numbers are zero-based or **%p** to indicate that the current source file should be used.

lsf, lsf- (Load or Unload Source File)

12/5/2018 • 2 minutes to read • Edit Online

The **lsf** and **lsf-** commands load or unload a source file.

```
lsf Filename
lsf- Filename
```

Parameters

Filename
Specifies the file to load or unload. If this file is not located in the directory where the debugger was opened from, you must include an absolute or relative path. The file name must follow Microsoft Windows file name conventions.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

The **lsf** command loads a source file.

The **lsf-** command unloads a source file. You can use this command to unload files that you previously loaded with **lsf** or automatically loaded source files. You cannot use **lsf-** to unload files that were loaded through WinDbg's File | Open Source File command or files that a WinDbg workspace loaded.

In CDB or KD, you can view source files in the Debugger Command window. In WinDbg, source files are loaded as new Source windows.

For more information about source files, source paths, and other ways to load source files, see Source Path.

See also

ls, lsa (List Source Lines)

lsc (List Current Source)

lsp (Set Number of Source Lines)

12/5/2018 • 2 minutes to read • Edit Online

The **lsp** command controls how many source lines are displayed while you step through or execute code or use the **ls and lsa commands**.

```
lsp [-a] LeadingLines TrailingLines
lsp [-a] TotalLines
lsp [-a]
```

Parameters

-a
Sets or displays the number of lines that **ls** and **lsa** show. If you omit **-a**, **lsp** sets or displays the number of lines that are shown while you step through and execute code.

LeadingLines
Specifies the number of lines to show before the current line.

TrailingLines
Specifies the number of lines to show after the current line.

TotalLines
Specifies the total number of lines to show. This number is divided evenly between leading and trailing lines. (If this number is odd, more trailing lines are displayed.)

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

When you use the **lsp** command together with no parameters, **lsp** displays the current leading line and trailing line values that you used while stepping. When you use this command together with only the **-a** parameter, **lsp** displays the values that you used while stepping and for the **ls and lsa commands**.

When you step through a program or break in after program execution, the previous **lsp** command determines the number of leading and trailing lines that are displayed. When you use **lsa**, the previous **lsp -a** command determines the number of leading and trailing lines that are displayed. When you use **ls**, all lines appear as a single block, so the previous **lsp -a** command determines the total number of lines that are displayed.

Additional Information

For more information about source debugging and related commands, see Debugging in Source Mode.

m (Move Memory)

12/5/2018 • 2 minutes to read • Edit Online

The **m** command copies the contents of memory from one location to another.

Do not confuse this command with the **~m (Resume Thread)** command.

```
m Range Address
```

Parameters

Range
Specifies the memory area to copy. For more information about the syntax of this parameter, see Address and Address Range Syntax.

Address
Specifies the starting address of the destination memory area. For more information about the syntax of this parameter, see Address and Address Range Syntax.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about memory manipulation and a description of other memory-related commands, see Reading and Writing Memory.

Remarks

The memory area that *Address* specifies can be part of the memory area that *Range* specifies. Overlapping moves are handled correctly.

n (Set Number Base)

12/5/2018 • 2 minutes to read • Edit Online

The **n** command sets the default number base (radix) to the specified value or displays the current number base.

Do not confuse this command with the **~n (Suspend Thread)** command.

```
n [Radix]
```

Parameters

Radix
Specifies the default number base that is used for numeric display and entry. You can use one of the following values.

VALUE	DESCRIPTION
8	Octal
10	Decimal
16	Hexadecimal

If you omit *Radix*, the current default number base is displayed.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

The current radix affects the input and output of MASM expressions. It does not affect the input or output of C++ expressions. For more information about these expressions, see Evaluating Expressions.

The default radix is set to 16 when the debugger is started.

In all MASM expressions, numeric values are interpreted as numbers in the current radix (16, 10, or 8). You can override the default radix by specifying the **0x** prefix (hexadecimal), the **0n** prefix (decimal), the **0t** prefix (octal), or the **0y** prefix (binary).

ob, ow, od (Output to Port)

12/5/2018 • 2 minutes to read • Edit Online

The **ob**, **ow**, and **od** commands send a byte, word, or double word to the selected port.

```
ob Address Value
ow Address Value
od Address Value
```

Parameters

Address
Specifies the address of the port.

Value
Specifies the hexadecimal value to write to the port.

Environment

Modes	Kernel mode only
Targets	Live debugging only
Platforms	x86-based only

Remarks

The **ob** command writes a single byte, the **ow** command writes a word, and the **od** command writes a double word.

Make sure that you do not send a word or a double-word to a port that does not support this size.

See also

ib, id, iw (Input from Port)

p (Step)

12/5/2018 • 2 minutes to read • Edit Online

The **p** command executes a single instruction or source line and optionally displays the resulting values of all registers and flags. When subroutine calls or interrupts occur, they are treated as a single step.

User-Mode

```
[~Thread] p[r] [= StartAddress] [Count] ["Command"]
```

Kernel-Mode

```
p[r] [= StartAddress] [Count] ["Command"]
```

Parameters

Thread
Specifies the threads to continue executing. All other threads are frozen. For more information about the syntax, see Thread Syntax. You can specify threads only in user mode.

r
Turns on and off the display of registers and flags. By default, the registers and flags are displayed. You can disable register display by using the **pr**, **tr**, or .prompt_allow -reg commands. All three of these commands control the same setting and you can use any of them to override any previous use of these commands.

You can also disable register display by using the l-os command. This setting is separate from the other three commands. To control which registers and flags are displayed, use the **rm (Register Mask)** command.

StartAddress
Specifies the address where execution should begin. If you do not use *StartAddress*, execution begins at the instruction that the instruction pointer points to. For more information about the syntax, see Address and Address Range Syntax.

Count
Specifies the number of instructions or source lines to step through before stopping. Each step is displayed as a separate action in the Debugger Command window. The default value is one.

Command
Specifies a debugger command to execute after the step is performed. This command is executed before the standard **p** results are displayed. If you also use *Count*, the specified command is executed after all stepping is complete (but before the results from the final step are displayed).

Environment

Modes	User mode, kernel mode
Targets	Live debugging only

Platforms	All

Additional Information

For more information about issuing the **p** command and an overview of related commands, see Controlling the Target.

Remarks

When you specify *Count*, each instruction is displayed as it is stepped through.

If the debugger encounters a **call** instruction or interrupt while stepping, the called subroutine will execute completely unless a breakpoint is encountered. Control is returned to the debugger at the next instruction after the call or interrupt.

Each step executes a single assembly instruction or a single source line, depending on whether the debugger is in assembly mode or source mode. Use the **l+t** and l-t commands or the buttons on the WinDbg toolbar to switch between these modes.

When you are quickly stepping many times in WinDbg, the debugging information windows are updated after each step. If this update causes slower response time, use **.suspend_ui (Suspend WinDbg Interface)** to temporarily suspend the refreshing of these windows.

pa (Step to Address)

12/5/2018 • 2 minutes to read • Edit Online

The **pa** command executes the program until the specified address is reached, displaying each step.

User-Mode

```
[~Thread] pa [r] [= StartAddress] StopAddress ["Command"]
```

Kernel-Mode

```
pa [r] [= StartAddress] StopAddress ["Command"]
```

Parameters

Thread
Specifies threads to continue executing. All other threads are frozen. For more information about the syntax, see Thread Syntax. You can specify threads only in user mode.

r
Turns on and off the display of registers and flags. By default, the registers and flags are displayed. You can disable register display by using the **par**, **pr**, **tr**, or .prompt_allow -reg commands. All of these commands control the same setting and you can use any of them to override any previous use of these commands.

You can also disable register display by using the l-os command. This setting is separate from the other three commands. To control which registers and flags are displayed, use the **rm (Register Mask)** command.

StartAddress
Specifies the address where the debugger begins execution. Otherwise, the debugger begins at the instruction that the instruction pointer points to. For more information about the syntax, see Address and Address Range Syntax.

StopAddress
Specifies the address where execution will stop. This address must match the exact address of an instruction.

Command
Specifies a debugger command to execute after the step is performed. This command is executed before the standard **pa** results are displayed. If you also use *StopAddress*, the specified command is executed after *StopAddress* is reached (but before the results from the final step are displayed).

Environment

Modes	User mode, kernel mode
Targets	Live debugging only
Platforms	All

Additional Information

For more information about related commands, see Controlling the Target.

Remarks

The **pa** command causes the target to begin executing. This execution continues until the specified instruction is reached or a breakpoint is encountered.

Note If you use this command in kernel mode, execution stops when an instruction is encountered at the specified virtual address in any virtual address space.

During this execution, all steps are displayed explicitly. Called functions are treated as a single unit. Therefore, the display of this command is similar to what you see if you execute **p (Step)** repeatedly until the program counter reaches the specified address.

For example, the following command explicitly steps through the target code until the return address of the current function is reached.

```
0:000> pa @$ra
```

The following example demonstrates using the **pa** command along with the **kb** command to display the stack trace:

```
0:000> pa 70b5d2f1 "kb"
```

pc (Step to Next Call)

12/5/2018 • 2 minutes to read • Edit Online

The **pc** command executes the program until a call instruction is reached.

User-Mode

```
[~Thread] pc [r] [= StartAddress] [Count]
```

Kernel-Mode

```
pc [r] [= StartAddress] [Count]
```

Parameters

Thread
Specifies threads to continue executing. All other threads are frozen. For more information about the syntax, see Thread Syntax. You can specify threads only in user mode.

r
Turns on and off the display of registers and flags. By default, the registers and flags are displayed. You can disable register display by using the **pcr**, **pr**, **tr**, or .prompt_allow -reg commands. All of these commands control the same setting and you can use any of them to override any previous use of these commands.

You can also disable register display by using the l-os command. This setting is separate from the other three commands. To control which registers and flags are displayed, use the **rm (Register Mask)** command.

StartAddress
Specifies the address where the debugger begins execution. Otherwise, the debugger begins at the instruction that the instruction pointer points to. For more information about the syntax, see Address and Address Range Syntax.

Count
Specifies the number of **call** instructions that the debugger must encounter for this command to stop. The default value is one.

Environment

Modes	User mode, kernel mode
Targets	Live debugging only
Platforms	All

Additional Information

For more information about related commands, see Controlling the Target.

Remarks

The **pc** command causes the target to begin executing. This execution continues until a **call** instruction is reached or a breakpoint is encountered.

If the program counter is already on a **call** instruction, the entire call is executed. After this call is returned, execution continues until another **call** is reached. This execution, rather than tracing, of the call is the only difference between **pc** and **tc (Trace to Next Call)**.

In source mode, you can associate one source line with multiple assembly instructions. The **pc** command does not stop at a **call** instruction that is associated with the current source line.

pct (Step to Next Call or Return)

12/5/2018 • 2 minutes to read • Edit Online

The **pct** command executes the program until it reaches a call instruction or a return instruction.

User-Mode

```
[~Thread] pct [r] [= StartAddress] [Count]
```

Kernel-Mode

```
pct [r] [= StartAddress] [Count]
```

Parameters

Thread
Specifies threads to continue executing. All other threads are frozen. For more information about the syntax, see Thread Syntax. You can specify threads only in user mode.

r
Turns on and off the display of registers and flags. By default, the registers and flags are displayed. You can disable register display through the **pctr**, **pr**, **tr**, or .prompt_allow -reg commands. All of these commands control the same setting and you can use any of them to override any previous use of these commands.

You can also disable register display by using the l-os command. This setting is separate from the other three commands. To control which registers and flags are displayed, use the **rm (Register Mask)** command.

StartAddress
Specifies the address where the debugger begins execution. Otherwise, the debugger begins at the instruction that the instruction pointer points to. For more information about the syntax, see Address and Address Range Syntax.

Count
Specifies the number of **call** or **return** instructions that must be encountered for this command to stop. The default value is one.

Environment

Modes	User mode, kernel mode
Targets	Live debugging only
Platforms	All

Additional Information

For more information about related commands, see Controlling the Target.

Remarks

The **pct** command causes the target to begin executing. This execution continues until a call or **return** instruction is reached or a breakpoint is encountered.

If the program counter is already on a **call** or **return** instruction, the entire call or return is executed. After this call or return is returned, execution continues until another **call** or **return** is reached. This execution, rather than tracing, of the call is the only difference between **pct** and **tct (Trace to Next Call or Return)**.

In source mode, you can associate one source line with multiple assembly instructions. The **pct** command does not stop at a **call** or **return** instruction that is associated with the current source line.

ph (Step to Next Branching Instruction)

12/5/2018 • 2 minutes to read • Edit Online

The **ph** command executes the program until any kind of branching instruction is reached, including conditional or unconditional branches, calls, returns, and system calls.

User-Mode

```
[~Thread] ph [r] [= StartAddress] [Count]
```

Kernel-Mode

```
ph [r] [= StartAddress] [Count]
```

Parameters

Thread
Specifies threads to continue executing. All other threads are frozen. For more information about the syntax, see Thread Syntax. You can specify threads only in user mode.

r
Turns on and off the display of registers and flags. By default, the registers and flags are displayed. You can disable register display by using the **phr**, **pr**, **tr**, or .prompt_allow -reg commands. All of these commands control the same setting and you can use any of them to override any previous use of these commands.

You can also disable register display by using the l-os command. This setting is separate from the other three commands. To control which registers and flags are displayed, use the **rm (Register Mask)** command.

StartAddress
Specifies the address where the debugger begins execution. Otherwise, the debugger begins at the instruction that the instruction pointer points to. For more information about the syntax, see Address and Address Range Syntax.

Count
Specifies the number of branching instructions that must be encountered for this command to stop. The default value is one.

Environment

Modes	User mode, kernel mode
Targets	Live debugging only
Platforms	All

Additional Information

For more information about related commands, see Controlling the Target.

Remarks

The **ph** command causes the target to begin executing. This execution continues until a branching instruction is reached or a breakpoint is encountered.

If the program counter is already on a branching instruction, the entire branching instruction is executed. After this branching instruction is returned, execution continues until another branching instruction is reached. This execution, rather than tracing, of the call is the only difference between **ph** and **th (Trace to Next Branching Instruction)**.

In source mode, you can associate one source line with multiple assembly instructions. The **ph** command does not stop at a branching instruction that is associated with the current source line.

pt (Step to Next Return)

12/5/2018 • 2 minutes to read • Edit Online

The **pt** command executes the program until a return instruction is reached.

User-Mode

```
[~Thread] pt [r] [= StartAddress] [Count] ["Command"]
```

Kernel-Mode

```
pt [r] [= StartAddress] [Count] ["Command"]
```

Parameters

Thread
Specifies threads to continue executing. All other threads are frozen. For more information about the syntax, see Thread Syntax. You can specify threads only in user mode.

r
Turns on and off the display of registers and flags. By default, the registers and flags are displayed. You can disable register display by using the **ptr**, **pr**, **tr**, or .prompt_allow -reg commands. All of these commands control the same setting and you can use any of them to override any previous use of these commands.

You can also disable register display by using the l-os command. This setting is separate from the other three commands. To control which registers and flags are displayed, use the **rm (Register Mask)** command.

StartAddress
Specifies the address where the debugger begins execution. Otherwise, the debugger begins at the instruction that the instruction pointer points to. For more information about the syntax, see Address and Address Range Syntax.

Count
Specifies the number of **return** instructions that must be encountered for this command to stop. The default value is one.

Command
Specifies a debugger command to execute after the step is performed. This command is executed before the standard **pt** results are displayed. If you also use *Count*, the specified command is executed after all stepping is complete (but before the results from the final step are displayed).

Environment

Modes	User mode, kernel mode
Targets	Live debugging only
Platforms	All

Additional Information

For more information about related commands, see Controlling the Target.

Remarks

The **pt** command causes the target to begin executing. This execution continues until a **return** instruction is reached or a breakpoint is encountered.

If the program counter is already on a **return** instruction, the entire return is executed. After this return is returned, execution continues until another **return** is reached. This execution, rather than tracing, of the call is the only difference between **pt** and **tt (Trace to Next Return)**.

In source mode, you can associate one source line with multiple assembly instructions. The **pt** command does not stop at a **return** instruction that is associated with the current source line.

The following example demonstrates using the **pt** command along with the **kb** command to display the stack trace:

```
0:000> pt "kb"
```

q, qq (Quit)

12/5/2018 • 2 minutes to read • Edit Online

The **q** and **qq** commands end the debugging session. (In CDB and KD, this command also exits the debugger itself. In WinDbg, this command returns the debugger to dormant mode.)

```
q
qq
```

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

In user-mode debugging, the **q** command ends the debugging session and closes the target application.

In kernel-mode debugging, the **q** command saves the log file and ends the debugging session. The target computer remains locked.

If this command does not work in KD, press **CTRL+R+ENTER** on the debugger keyboard, and then retry the **q** command. If this action does not work, you must use CTRL+B+ENTER to exit the debugger.

The **qq** command behaves exactly like the **q** command, unless you are performing remote debugging. During remote debugging, the **q** command has no effect, but the **qq** command ends the debugging server. For more information about this effect, see Remote Debugging Through the Debugger.

qd (Quit and Detach)

12/5/2018 • 2 minutes to read • Edit Online

The **qd** command ends the debugging session and leaves any user-mode target application running. (In CDB and KD, this command also exits the debugger itself. In WinDbg, this command returns the debugger to dormant mode.)

```
qd
```

Environment

Modes	User mode only
Targets	Live debugging only
Platforms	All

Remarks

The **qd** command detaches from a target application and ends the debugging session, leaving the target still running. However, this command is supported only on Microsoft Windows XP and later versions of Windows. On Windows 2000, **qd** generates a warning message and has no effect.

When you are performing remote debugging through the debugger, you cannot use the **qd** command from a debugging client.

r (Registers)

12/5/2018 • 5 minutes to read • Edit Online

The **r** command displays or modifies registers, floating-point registers, flags, pseudo-registers, and fixed-name aliases.

User-Mode

```
[~Thread] r[M Mask|F|X|?] [ Register[:[Num]Type] [= [Value]] ]
r.
```

Kernel-Mode

```
[Processor] r[M Mask|F|X|Y|YI|?] [ Register[:[Num]Type] [= [Value]] ]
r.
```

Parameters

Processor
Specifies the processor that the registers are read from. The default value is zero. If you specify *Processor*, you cannot include the *Register* parameter--all registers are displayed. For more information about the syntax, see Multiprocessor Syntax. You can specify processors only in kernel mode.

Thread
Specifies the thread that the registers are read from. If you do not specify a thread, the current thread is used. For more information about the syntax, see Thread Syntax. You can specify threads only in user mode.

M *Mask*
Specifies the mask to use when the debugger displays the registers. The "M" must be an uppercase letter. *Mask* is a sum of bits that indicate something about the register display. The meaning of the bits depends on the processor and the mode (see the tables in the following Remarks section for more information). If you omit **M**, the default mask is used. You can set or display the default mask by using the **Rm (Register Mask)** command.

F
Displays the floating-point registers. The "F" must be an uppercase letter. This option is equivalent to **M 0x4**.

X
Displays the SSE XMM registers. This option is equivalent to **M 0x40**.

Y
Displays the AVX YMM registers. This option is equivalent to **M 0x200**.

YI
Displays the AVX YMM integer registers. This option is equivalent to **M 0x400**.

Z
Displays the AVX-512 YMM registers (zmm0-zmm31) in floating point format.

ZI
Displays the AVX-512 YMM registers (zmm0-zmm31) in integer format.

K
Display the AVX-512 Opmask predicate registers (K0-K7).

?
(Pseudo-register assignment only) Causes the pseudo-register to acquire typed information. Any type is permitted. For more information about the **r?** syntax, see Debugger Command Program Examples.

Register
Specifies the register, flag, pseudo-register, or fixed-name alias to display or modify. You must not precede this parameter with at (**@**) sign. For more information about the syntax, see Register Syntax.

Num
Specifies the number of elements to display. If you omit this parameter but you include *Type*, the full register length is displayed.

Type
Specifies the data format to display each register element in. You can use *Type* only with 64-bit and 128-bit vector registers. You can specify multiple types.

You can specify one or more of the following values.

TYPE	DISPLAY FORMAT
ib	Signed byte
ub	Unsigned byte
iw	Signed word
uw	Unsigned word
id	Signed DWORD
ud	Unsigned DWORD
iq	Signed quad-word
uq	Unsigned quad-word
f	32-bit floating-point
d	64-bit floating-point

Value
Specifies the value to assign to the register. For more information about the syntax, see Numerical Expression Syntax.

.
Displays the registers used in the current instruction. If no registers are used, no output is displayed.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about the register context and other context settings, see Changing Contexts.

Remarks

If you do not specify *Register*, the **r** command displays all the non-floating-point registers, and the **rF** command displays all the floating-point registers. You can change this behavior by using the **rm (Register Mask)** command.

If you specify *Register* but you omit the equal sign (=) and the *Value* parameter, the command displays the current value of the register.

If you specify *Register* and an equal sign (=) but you omit *Value*, the command displays the current value of the register and prompts for a new value.

If you specify *Register*, the equal sign (=), and *Value*, the command changes the register to contain the value. (If *quiet mode* is active, you can omit the equal sign. You can turn on quiet mode by using the **sq (Set Quiet Mode)** command. In kernel mode, you can also turn on quiet mode by using the KDQUIET environment variable.)

You can specify multiple registers, separated by commas.

In user mode, the **r** command displays registers that are associated with the current thread. For more information about the threads, see Controlling Processes and Threads.

In kernel mode, the **r** command displays registers that are associated with the current *register context*. You can set the register context to match a specific thread, context record, or trap frame. Only the most important registers for the specified register context are actually displayed, and you cannot change their values. For more information about register context, see Register Context.

When you specify a floating-point register by name, the **F** option is not required. When you specify a single floating-point register, the raw hexadecimal value is displayed in addition to the decimal value.

The following *Mask* bits are supported for an x86-based processor or an x64-based processor.

BIT	VALUE	DESCRIPTION
0 1	0x1 0x2	Displays the basic integer registers. (Setting one or both of these bits has the same effect.)
2	0x4	Displays the floating-point registers.

BIT	VALUE	DESCRIPTION
3	0x8	Displays the segment registers.
4	0x10	Displays the MMX registers.
5	0x20	Displays the debug registers. In kernel mode, setting this bit also displays the CR4 register.
6	0x40	Displays the SSE XMM registers.
7	0x80	(Kernel mode only) Displays the control registers, for example CR0, CR2, CR3 and CR8.
8	0x100	(Kernel mode only) Displays the descriptor and task state registers.
9	0x200	Displays the AVX YMM registers in floating point.
10	0x400	Displays the AVX YMM registers in decimal integers.
11	0x800	Displays the AVX XMM registers in decimal integers.

The following code examples show **r** commands for an x86-based processor.

In kernel mode, the following command shows the registers for processor 2.

```
1: kd> 2r
```

In user mode, the following command shows the registers for thread 2.

```
0:000> ~2 r
```

In user mode, the following command displays all of the **eax** registers that are associated with all threads (in thread index order).

```
0:000> ~* r eax
```

The following command sets the **eax** register for the current thread to 0x000000FF.

```
0:000> r eax=0x000000FF
```

The following command sets the **st0** register to 1.234e+10 (the **F** is optional).

```
0:000> rF st0=1.234e+10
```

The following command displays the zero flag.

```
0:000> r zf
```

The following command displays the **xmm0** register as 16 unsigned bytes and then displays the full contents of the **xmm1** register in double-precision floating-point format.

```
0:000> r xmm0:16ub, xmm1:d
```

If the current syntax is C++, you must precede registers by an at sign (**@**). Therefore, you could use the following command to copy the **ebx** register to the **eax** register.

```
0:000> r eax = @ebx
```

The following command displays pseudo-registers in the same way that the **r** command displays registers.

```
0:000> r $teb
```

You can also use the **r** command to create *fixed-name aliases*. These aliases are not registers or pseudo-registers, even though they are associated with the **r** command. For more information about these aliases, see Using Aliases.

Here is an example of the **r.** command on an x86-based processor. The last entry of the call stack precedes the command itself.

```
01004af3 8bec            mov     ebp,esp
0:000> r.
ebp=0006ffc0  esp=0006ff7c
```

rdmsr (Read MSR)

12/5/2018 • 2 minutes to read • Edit Online

The **rdmsr** command reads a Model-Specific Register (MSR) value from the specified address.

```
rdmsr Address
```

Parameters

Address
Specifies the address of the MSR.

Environment

Modes	Kernel mode only
Targets	Live debugging only
Platforms	All

Remarks

The **rdmsr** command can display MSR's on x86-based, Itanium-based, and x64-based platforms. The MSR definitions are platform-specific.

See also

wrmsr (Write MSR)

rm (Register Mask)

12/5/2018 • 2 minutes to read • Edit Online

The **rm** command modifies or displays the register display mask. This mask controls how registers are displayed by the **r (Registers)** command.

```
rm
rm ?
rm Mask
```

Parameters

?
Displays a list of possible *Mask* bits.

Mask
Specifies the mask to use when the debugger displays the registers. *Mask* is a sum of bits that indicate something about the register display. The meaning of the bits depends on the processor and the mode. For more information; see the tables in the following Remarks section.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

The "m" in the command name must be a lowercase letter.

If you use **rm** with no parameters, the current value is displayed, along with an explanation about its bits.

To display the basic integer registers, you must set bit 0 (0x1) or bit 1 (0x2). By default, 0x1 is set for 32-bit targets and 0x2 is set for 64-bit targets. You cannot set these two bits at the same time--if you try to set both bits, 0x2 overrides 0x1.

You can override the default mask by using the **r (Registers)** command together with the **M** option.

The following *Mask* bits are supported for an x86-based processor or an x64-based processor.

BIT	VALUE	DESCRIPTION
0 1	0x1 0x2	Displays the basic integer registers. (Setting one or both of these bits has the same effect.)

BIT	VALUE	DESCRIPTION
2	0x4	Displays the floating-point registers.
3	0x8	Displays the segment registers.
4	0x10	Displays the MMX registers.
5	0x20	Displays the debug registers. In kernel mode, setting this bit also displays the CR4 register.
6	0x40	Displays the SSE XMM registers.
7	0x80	(Kernel mode only) Displays the control registers, for example CR0, CR2, CR3 and CR8.
8	0x100	(Kernel mode only) Displays the descriptor and task state registers.
9	0x200	Displays the AVX YMM registers in floating point.
10	0x400	Displays the AVX YMM registers in decimal integers.
11	0x800	Displays the AVX XMM registers in decimal integers.
12	0x1000	Displays the AVX-512 zmm0-zmm31 registers in floating point format.
13	0x2000	Displays the AVX-512 zm00-zmm31 registers in integer format.
14	0x4000	Displays the AVX-512 k0-k7 registers.

Examples

Enable the integer state and segment registers.

```
0: kd> rm 0x00a
0: kd> rm
Register output mask is a:
       2 - Integer state (64-bit)
       8 - Segment registers
```

Enable 0x1000 (Displays the AVX-512 zmm0-zmm31 registers in floating point format).

```
0: kd> rm 0x100a
0: kd> rm
Register output mask is 100a:
       2 - Integer state (64-bit)
       8 - Segment registers
    1000 - AVX-512 ZMM registers
```

Enable mask 0x2000 (Displays the AVX-512 zmm00-zmm31 registers in integer format).

```
0: kd> rm 0x200a
0: kd> rm
Register output mask is 200a:
       2 - Integer state (64-bit)
       8 - Segment registers
    2000 - AVX-512 ZMM Integer registers
```

Enable all AVX-512 register masks:

```
0: kd> rm 0x700a
0: kd> rm
Register output mask is 700a:
       2 - Integer state (64-bit)
       8 - Segment registers
    1000 - AVX-512 ZMM registers
    2000 - AVX-512 ZMM Integer registers
    4000 - AVX-512 Opmask registers
```

If you try and set a register mask on hardware that does not support it, the invalid bits of the register mask will be ignored.

```
kd> rm 0x100a
Ignored invalid bits 1000
kd> rm
Register output mask is a:
      2 - Integer state (64-bit)
       8 - Segment registers
```

s (Search Memory)

2/22/2019 • 6 minutes to read • Edit Online

The **s** command searches through memory to find a specific byte pattern.

Do not confuse this command with the **~s (Change Current Processor)**, **~s (Set Current Thread)**, **|s (Set Current Process)**, or **||s (Set Current System)** commands.

```
s [-[[Flags]Type]] Range Pattern
s -[[Flags]]v Range Object
s -[[Flags]]sa Range
s -[[Flags]]su Range
```

Parameters

[*Flags*]
Specifies one or more search options. Each flag is a single letter. You must enclose the flags in a single set of brackets ([]). You cannot add spaces between the brackets, except between **n** or **l** and its argument. For example, if you want to specify the **s** and **w** options, use the command s -[sw]Type Range Pattern.

You can specify one or more of the following flags:

s
Saves all of the results of the current search. You can use these results to repeat the search later.

r
Restricts the current search to the results from the last saved search. You cannot use the **s** and **r** flags in the same command. When you use **r**, the value of *Range* is ignored, and the debugger searches only those hits that were saved by the previous **s** command.

n *Hits*
Specifies the number of hits to save when you use the **s** flag. The default value is 1024 hits. If you use **n** together with other flags, **n** must be the last flag, followed by its *Hits* argument. The space between **n** and *Hits* is optional, but you cannot add any other spaces within the brackets. If any later search that uses the **s** flag discovers more than the specified number of hits, the **Overflow error** message is displayed to notify you that not all hits are being saved.

l *Length*
Causes a search for arbitrary ASCII or Unicode strings to return only strings that are at least *Length* characters long. The default length is 3. This value affects only searches that use the **-sa** or **-su** flags.

w
Searches only writeable memory regions. You must enclose the "w" in brackets.

1
Displays only the addresses of search matches in the search output. This option is useful if you are using the **.foreach** token to pipe the command output into another command's input.

Type
Specifies the memory type to search for. Add a hyphen (-) in front of *Type* . You can use one of the following *Type* values.

TYPE	DESCRIPTION
b	Byte (8 bits)
w	WORD (16 bits)
d	DWORD (32 bits)
q	QWORD (64 bits)
a	ASCII string (not necessarily a null-terminated string)
u	Unicode string (not necessarily a null-terminated string)

If you omit *Type*, byte values are used. However, if you use *Flags*, you cannot omit *Type*.

sa
Searches for any memory that contains printable ASCII strings. Use the **l** *Length* flag to specify a minimum length of such strings. The default minimum length is 3 characters.

su
Searches for any memory that contains printable Unicode strings. Use the **l** *Length* flag to specify a minimum length of such strings. The default minimum length is 3 characters.

Range
Specifies the memory area to search through. This range cannot be more than 256 MB long unless you use the **L?** syntax. For more information about this syntax, see Address and Address Range Syntax.

Pattern
Specifies one or more values to search for. By default, these values are byte values. You can specify different types of memory in *Type*. If you specify a WORD, DWORD, or QWORD value, depending on the other options selected, you may need to enclose the search pattern in single quotation marks (for example, **'H'**).

```
0:000> s -d @rsp L1000000 'H'
0000003f`ff07ec00  00000048 00000000 500c0163 00000000  H.......c..P....
0000003f`ff07ec50  00000048 00000000 00000080 00000000  H...............
0000003f`ff07efc0  00000048 00000000 400c0060 00000000  H.......`..@....
```

If you specify a string, using the ascii type, enclose it in double quotation marks (for example, **"B7"**).

```
0:000> s -a @rsp L10000000 "B7"
0000003f`ff07ef0a  42 37 ff 7f 00 00 90 38-4e c2 6c 01 00 00 7d 26  B7.....8N.l...}&
0000003f`ff0ff322  42 37 ff 7f 00 00 f8 5d-42 37 ff 7f 00 00 20 41  B7.....]B7.... A
0000003f`ff0ff32a  42 37 ff 7f 00 00 20 41-42 37 ff 7f 00 00 98 59  B7.... AB7.....Y
```

-v
Searches for objects of the same type as the specified *Object*.

Object
Specifies the address of an object or the address of a pointer to an object. The debugger then searches for objects of the same type as the object that *Object* specifies.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about memory manipulation and a description of other memory-related commands, see Reading and Writing Memory.

Remarks

If the debugger finds the byte pattern that you specify, the debugger displays the first memory address in the *Range* memory area where the pattern was found. The debugger displays an excerpt of memory that begins at that location in a format that matches the specified *Type* memory type. If *Type* is **a** or **u**, the memory contents and the corresponding ASCII or Unicode characters are displayed.

You must specify the *Pattern* parameter as a series of bytes, unless you specify a different *Type* value. You can enter byte values as numeric or ASCII characters:

- Numeric values are interpreted as numbers in the current radix (16, 10, or 8). To change the default radix, use the **n (Set Number Base)** command. You can override the default radix by specifying the **0x** prefix (hexadecimal), the **0n** prefix (decimal), the **0t** prefix (octal), or the **0y** prefix (binary). **Note** The default radix behaves differently when you use C++ expressions. For more information about these expressions and the radix, see Evaluating Expressions.
- You must enclose ASCII characters in single straight quotation marks. You cannot use C-style escape characters (such as '\0' or '\n').

If you specify multiple bytes, you must separate them by spaces.

The **s-a** and **s-u** commands search for specified ASCII and Unicode strings, respectively. These strings do not have to be null-terminated.

The **s-sa** and **s-su** commands search for unspecified ASCII and Unicode strings. These are useful if you are checking a range of memory to see whether it contains any printable characters. The flags options allow you to specify a minimum length of string to find.

Example: The following command finds ASCII strings that are of length >=3 in the range beginning at 0000000140000000 and ending 400 bytes later.

```
s-sa 0000000140000000 L400
```

The following command finds ASCII strings that are of length >=4 in the range beginning at 0000000140000000 and ending 400 bytes later

```
s -[l4]sa 0000000140000000 L400
```

The following command does the same thing, but it limits the search to writeable memory regions.

```
s -[wl4]sa 0000000140000000 L400
```

The following command does the same thing, but displays only the address of the match, rather than the address and the value.

```
s -[1wl4]sa 0000000140000000 L400
```

The **s-v** command searches for objects of the same data type as the *Object* object. You can use this command only if the desired object is a C++ class or another object that is associated with virtual function tables (Vtables). The **s-v** command searches the *Range* memory area for the addresses of this class's Vtables. If multiple Vtables exist in this class, the search algorithm looks for all of these pointer values, separated by the proper number of bytes. If any matches are found, the debugger returns the base address of the object and full information about this object--similar to the output of the **dt (Display Type)** command.

Example: Assume the current radix is 16. The following three command all do the same thing: search memory locations 0012FF40 through 0012FF5F for "Hello".

```
0:000> s 0012ff40 L20 'H' 'e' 'l' 'l' 'o'
```

```
0:000> s 0012ff40 L20 48 65 6c 6c 6f
```

```
0:000> s -a 0012ff40 L20 "Hello"
```

These commands locate each appearance of "Hello" and return the address of each such pattern--that is, the address of the letter "H".

The debugger returns only patterns that are completely contained in the search range. Overlapping patterns are found correctly. (In other words, the pattern "QQQ" is found three times in "QQQQQ".)

The following example shows a search that uses the *Type* parameter. This command searches memory locations 0012FF40 through 0012FF5F for the double-word 'VUTS':

```
0:000> s -d 0012ff40 L20 'VUTS'
```

On little-endian computers, 'VUTS' is the same as the byte pattern 'S' 'T' 'U' 'V'. However, searches for WORDs, DWORDs, and QWORDs return only results that are correctly byte-aligned.

so (Set Kernel Debugging Options)

12/5/2018 • 2 minutes to read • Edit Online

The **so** command sets or displays the kernel debugging options.

```
so [Options]
```

Parameters

Options
One or more of the following options:

NOEXTWARNING
Does not issue a warning when the debugger cannot find an extension command.

NOVERSIONCHECK
Does not check the version of debugger extension DLLs.

If you omit *Options*, the current options are displayed.

Environment

Modes	Kernel mode only
Targets	Live, crash dump
Platforms	All

Remarks

You can also set kernel debugging options using the _NT_DEBUG_OPTIONS environment variable.

sq (Set Quiet Mode)

12/5/2018 • 2 minutes to read • Edit Online

The **sq** command turns quiet mode on or off.

```
sq
sq{e|d}
```

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

The **sqe** command turns quiet mode on, and the **sqd** command turns it off. The **sq** command turns on and off quiet mode. Each of these commands also displays the new quiet mode status.

You can set quiet mode in KD or kernel-mode WinDbg by using the KDQUIET environment variable. (Note that quiet mode exists in both user-mode and kernel-mode debugging, but the KDQUIET environment variable is only recognized in kernel mode.)

Quiet mode has three distinct effects:

- The debugger does not display messages every time that an extension DLL is loaded or unloaded.
- The **r (Registers)** command no longer requires an equal sign (=) in its syntax.
- When you break into a target computer while kernel debugging, the long warning message is suppressed.

Do not confuse quiet mode with the effects of the **-myob** command-line option (in CDB and KD) or the **-Q command-line option** (in WinDbg).

ss (Set Symbol Suffix)

12/5/2018 • 2 minutes to read • Edit Online

The **ss** command sets or displays the current suffix value that is used for symbol matching in numeric expressions.

```
ss [a|w|n]
```

Parameters

a

Specifies that the symbol suffix should be "A", matching many ASCII symbols.

w

Specifies that the symbol suffix should be "W", matching many Unicode symbols.

n

Specifies that the debugger should not use a symbol suffix. (This parameter is the default behavior.)

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about symbol matching, see Symbol Syntax and Symbol Matching.

Remarks

If you specify the **ss** command together with no parameters, the current state of the suffix value is displayed.

sx, sxd, sxe, sxi, sxn, sxr, sx- (Set Exceptions)

10/8/2019 • 4 minutes to read • Edit Online

The **sx** commands control the action that the debugger takes when an exception occurs in the application that is being debugged, or when certain events occur.

```
sx

sx{e|d|i|n} [-c "Cmd1"] [-c2 "Cmd2"] [-h] {Exception|Event|*}

sx- [-c "Cmd1"] [-c2 "Cmd2"] {Exception|Event|*}

sxr
```

Parameters

-c "*Cmd1***"**
Specifies a command that is executed if the exception or event occurs. This command is executed when the first chance to handle this exception occurs, regardless of whether this exception breaks into the debugger. You must enclose the *Cmd1* string in quotation marks. This string can include multiple commands, separated by semicolons. The space between the -c and the quoted command string is optional.

-c2"*Cmd2***"**
Specifies a command that is executed if the exception or event occurs and is not handled on the first chance. This command is executed when the second chance to handle this exception occurs, regardless of whether this exception breaks into the debugger. You must enclose the *Cmd2* string in quotation marks. This string can include multiple commands, separated by semicolons. The space between the -c2 and the quoted command string is optional.

-h Changes the specified event's handling status instead of its break status. If *Event* is **cc**, **hc**, **bpec**, or **ssec**, you do not have to use the **-h** option.

Exception Specifies the exception number that the command acts on, in the current radix.

Event Specifies the event that the command acts on. These events are identified by short abbreviations. For a list of the events, see Controlling Exceptions and Events.

***** Affects all exceptions that are not otherwise explicitly named for **sx**. For a list of explicitly named exceptions, see Controlling Exceptions and Events.

Environment

Modes	User mode, kernel mode
Targets	Live debugging only
Platforms	All

Additional Information

For more information about break status and handling status, descriptions of all event codes, a list of the default

status for all events, and other methods of controlling this status, see Controlling Exceptions and Events.

Remarks

The **sx** command displays the list of exceptions for the current process and the list of all nonexception events and displays the default behavior of the debugger for each exception and event.

The **sxe**, **sxd**, **sxn**, and **sxi** commands control the debugger settings for each exception and event.

The **sxr** command resets all of the exception and event filter states to the default settings. Commands are cleared, break and continue options are reset to their default settings, and so on.

The **sx-** command does not change the handling status or the break status of the specified exception or event. This command can be used if you wish to change the first-chance command or second-chance command associated with a specific event, but do not wish to change anything else.

If you include the **-h** option (or if the **cc**, **hc**, **bpec**, or **ssec** events are specified), the **sxe**, **sxd**, **sxn**, and **sxi** commands control the handling status of the exception or event. In all other cases, these commands control the break status of the exception or event.

When you are setting the break status, these commands have the following effects.

COMMAND	STATUS NAME	DESCRIPTION
sxe	**Break** **(Enabled)**	When this exception occurs, the target immediately breaks into the debugger before any other error handlers are activated. This kind of handling is called *first chance* handling.
sxd	**Second chance break** **(Disabled)**	The debugger does not break for a first-chance exception of this type (although a message is displayed). If other error handlers do not address this exception, execution stops and the target breaks into the debugger. This kind of handling is called *second chance* handling.
sxn	**Output** **(Notify)**	When this exception occurs, the target application does not break into the debugger at all. However, a message is displayed that notifies the user of this exception.
sxi	**Ignore**	When this exception occurs, the target application does not break into the debugger at all, and no message is displayed.

When you are setting the handling status, these commands have the following effects:

COMMAND	STATUS NAME	DESCRIPTION
sxe	**Handled**	The event is considered handled when execution resumes.
sxd,sxn,sxi	**Not Handled**	The event is considered not handled when execution resumes.

You can use the **-h** option together with exceptions, not events. Using this option with **ch**, **bpe**, or **sse** sets the handling status for **hc**, **bpec**, or **ssec**, respectively. If you use the -h option with any other event, it has no effect.

Using the **-c** or **-c2** options with **hc**, **bpec**, or **ssec** associates the specified commands with **ch**, **bpe**, or **sse**, respectively.

In the following example, the **sxe** command is used to set the break status of access violation events to break on the first chance, and to set the first-chance command that will be executed at that point to **r eax**. Then the **sx-** command is used to alter the first-chance command to **r ebx**, without changing the handling status. Finally, a portion of the **sx** output is shown, indicating the current settings for access violation events:

```
0:000> sxe -c "r eax" av

0:000> sx- -c "r ebx" av

0:000> sx
 av - Access violation - break - not handled
       Command: "r ebx"
  . . .
```

t (Trace)

12/5/2018 • 2 minutes to read • Edit Online

The **t** command executes a single instruction or source line and optionally displays the resulting values of all registers and flags. When subroutine calls or interrupts occur, each of their steps is also traced.

User-Mode

```
[~Thread] t [r] [= StartAddress] [Count] ["Command"]
```

Kernel-Mode

```
t [r] [= StartAddress] [Count] ["Command"]
```

Parameters

Thread
Specifies threads to thaw. All other threads are frozen. For more information about this syntax, see Thread Syntax. You can specify threads only in user mode.

r
Turns on and off the display of registers and flags. By default, the registers and flags are displayed. You can disable register display by using the **pr**, **tr**, or .prompt_allow -reg commands. All three of these commands control the same setting and you can use any of them to override any previous use of these commands.

You can also disable register display by using the l-os command. This setting is separate from the other three commands. To control which registers and flags are displayed, use the **rm (Register Mask)** command.

StartAddress
Specifies the address where execution should begin. If you do not use *StartAddress*, execution begins at the instruction that the instruction pointer points to. For more information about the syntax, see Address and Address Range Syntax.

Count
Specifies the number of instructions or source lines to trace through before stopping. Each step is displayed as a separate action in the Debugger Command window. The default value is one.

Command
Specifies a debugger command to execute after the trace is performed. This command is executed before the standard **t** results are displayed. If you also use *Count*, this command is executed after all tracing is complete (but before the results from the final trace are displayed).

Environment

Modes	User mode, kernel mode
Targets	Live debugging only

Platforms	All

Additional Information

For more information about how to issue the **t** command and an overview of related commands, see Controlling the Target.

Remarks

When you specify *Count*, each instruction is displayed as it is stepped through.

Each trace executes a single assembly instruction or a single source line, depending on whether the debugger is in assembly mode or source mode. Use the **l+t** and l-t commands or the buttons on the WinDbg toolbar to switch between these modes.

If you want to trace most function calls but skip certain calls, you can use **.step_filter (Set Step Filter)** to indicate which calls to step over.

You can use the **t** command to trace instructions in ROM.

When you are quickly tracing many times in WinDbg, the debugging information windows are updated after each trace. If this update causes slower response time, use **.suspend_ui (Suspend WinDbg Interface)** to temporarily suspend the updating of these windows.

ta (Trace to Address)

12/5/2018 • 2 minutes to read • Edit Online

The **ta** command executes the program until the specified address is reached, displaying each step (including steps within called functions).

User-Mode

```
[~Thread] ta [r] [= StartAddress] StopAddress
```

Kernel-Mode

```
ta [r] [= StartAddress] StopAddress
```

Parameters

Thread
Specifies threads to continue executing. All other threads are frozen. For more information about the syntax, see Thread Syntax. You can specify threads only in user mode.

r
Turns on and off the display of registers and flags. By default, the registers and flags are displayed. You can disable register display by using the **tar**, **pr**, **tr**, or .prompt_allow -reg commands. All of these commands control the same setting and use of any of them overrides any previous use of these commands.

You can also disable register display by using the l-os command. This setting is separate from the other four commands. To control which registers and flags are displayed, use the **rm (Register Mask)** command.

StartAddress
Specifies the address where the debugger begins execution. If you do not use *StartAddress*, execution begins at the instruction that the instruction pointer points to. For more information about the syntax, see Address and Address Range Syntax.

StopAddress
Specifies the address at which execution stops. This address must match the exact address of an instruction.

Environment

Modes	User mode, kernel mode
Targets	Live debugging only
Platforms	All

Additional Information

For more information about related commands, see Controlling the Target.

Remarks

The **ta** command causes the target to begin executing. This execution continues until the specified instruction is reached or a breakpoint is encountered.

Note If you use the **ta** command in kernel mode, execution stops when an instruction is encountered at the specified virtual address in any virtual address space.

During this execution, all steps are displayed explicitly. If a function is called, the debugger also traces through that function. Therefore, the display of this command resembles what you see if you executed **t (Trace)** repeatedly until the program counter reached the specified address.

For example, the following command explicitly traces through the target code until the return address of the current function is reached.

```
0:000> ta @$ra
```

tb (Trace to Next Branch)

12/5/2018 • 2 minutes to read • Edit Online

The **tb** command executes the program until a branch instruction is reached.

```
tb [r] [= StartAddress] [Count]
```

Parameters

r

Turns on and off the display of registers and flags. By default, the registers and flags are displayed. You can disable register display by using the **tbr**, **pr**, **tr**, or .prompt_allow -reg commands. All of these commands control the same setting and you can use any of them to override any previous use of these commands.

You can also disable register display by using the l-os command. This setting is separate from the other four commands. To control which registers and flags are displayed, use the **rm (Register Mask)** command.

StartAddress

Specifies the address where the debugger starts execution. If you do not use *StartAddress*, execution begins at the instruction that the instruction pointer points to. For more information about the syntax, see Address and Address Range Syntax.

Count

Specifies the number of branches to allow. Every time that a branch is encountered, the instruction address and the instruction are displayed. If you omit *Count*, the default number is 1.

Environment

Modes	**x86-based:** Kernel mode only **Itanium-based:** User mode, kernel mode **x64-based:** User mode, kernel mode
Targets	Live debugging only
Platforms	x86-based (GenuineIntel processor family 6 and later), Itanium-based, x64-based

Additional Information

For more information about related commands, see Controlling the Target.

Remarks

The **tb** command causes the target to begin executing. This execution continues until a branch command is reached.

Execution stops at any branch command that is to be taken. This stopping of execution is always based on the *disassembly* code, even when the debugger is in source mode.

Branch instructions include calls, returns, jumps, counted loops, and while loops. If the debugger encounters an unconditional branch, or a conditional branch for which the condition is true, execution stops. If the debugger

encounters a conditional branch whose condition is false, execution continues.

When execution stops, the address of the branch instruction and any associated symbols are displayed. This information is followed by an arrow and then the address and instructions of the new program counter location.

The **tb** command works only on the current processor. If you use **tb** on a multiprocessor system, execution stops when a branch command is reached or when another processor's event occurs, whichever comes first.

Usually, branch tracing is enabled after the processor control block (PRCB) has been initialized. (The PRCB is initialized early in the boot process.) However, if you have to use the **tb** command before this point, you can use **.force_tb (Forcibly Allow Branch Tracing)** to enable branch tracing earlier. Use the **.force_tb** command cautiously, because it can corrupt your processor state.

tc (Trace to Next Call)

12/5/2018 • 2 minutes to read • Edit Online

The **tc** command executes the program until a call instruction is reached.

User-Mode

```
[~Thread] tc [r] [= StartAddress] [Count]
```

Kernel-Mode

```
tc [r] [= StartAddress] [Count]
```

Parameters

Thread
Specifies threads to continue executing. All other threads are frozen. For more information about the syntax, see Thread Syntax. You can specify threads only in user mode.

r
Turns on and off the display of registers and flags. By default, the registers and flags are displayed. You can disable register display by using the **tcr**, **pr**, **tr**, or .prompt_allow -reg commands. All of these commands control the same setting and you can use any of them to override any previous use of these commands.

You can also disable register display by using the l-os command. This setting is separate from the other four commands. To control which registers and flags are displayed, use the **rm (Register Mask)** command.

StartAddress
Specifies the address where the debugger begins execution. If you do not use *StartAddress*, execution begins at the instruction that the instruction pointer points to. For more information about the syntax, see Address and Address Range Syntax.

Count
Specifies the number of **call** instructions that the debugger must encounter for the **tc** command to end. The default value is one.

Environment

Modes	User mode, kernel mode
Targets	Live debugging only
Platforms	All

Additional Information

For more information about related commands, see Controlling the Target.

Remarks

The **tc** command causes the target to begin executing. This execution continues until the debugger reaches a **call** instruction or encounters a breakpoint.

If the program counter is already on a **call** instruction, the debugger traces into the call and continues executing until it encounters another **call**. This tracing, rather than execution, of the call is the only difference between **tc** and **pc (Step to Next Call)**.

In source mode, you can associate one source line with multiple assembly instructions. This command does not stop at a **call** instruction that is associated with the current source line.

tct (Trace to Next Call or Return)

12/5/2018 • 2 minutes to read • Edit Online

The **tct** command executes the program until it reaches a call instruction or return instruction.

User-Mode

```
[~Thread] tct [r] [= StartAddress] [Count]
```

Kernel-Mode

```
tct [r] [= StartAddress] [Count]
```

Parameters

Thread
Specifies threads to continue executing. All other threads are frozen. For more information about the syntax, see Thread Syntax. You can specify threads only in user mode.

r
Turns on and off the display of registers and flags. By default, the registers and flags are displayed. You can disable register display by using the **tctr**, **pr**, **tr**, or .prompt_allow -reg commands. All of these commands control the same setting and you can use any of them to override any previous use of these commands.

You can also disable register display by using the l-os command. This setting is separate from the other four commands. To control which registers and flags are displayed, use the **rm (Register Mask)** command.

StartAddress
Specifies the address where the debugger begins execution. If you do not use *StartAddress*, execution begins at the instruction that the instruction pointer points to. For more information about the syntax, see Address and Address Range Syntax.

Count
Specifies the number of **call** or **return** instructions that the debugger must encounter for the **tct** command to end. The default value is one.

Environment

Modes	User mode, kernel mode
Targets	Live debugging only
Platforms	All

Additional Information

For more information about related commands, see Controlling the Target.

Remarks

The **tct** command causes the target to begin executing. This execution continues until the debugger reaches a **call** or **return** instruction or encounters a breakpoint.

If the program counter is already on a **call** or **return** instruction, the debugger traces into the call or return and continues executing until it encounters another **call** or **return**. This tracing, rather than execution, of the call is the only difference between **tct** and **pct (Step to Next Call or Return)**.

In source mode, you can associate one source line with multiple assembly instructions. This command does not stop at a **call** or **return** instruction that is associated with the current source line.

th (Trace to Next Branching Instruction)

12/5/2018 • 2 minutes to read • Edit Online

The **th** command executes the program until it reaches any kind of branching instruction, including conditional or unconditional branches, calls, returns, and system calls.

User-Mode

```
[~Thread] th [r] [= StartAddress] [Count]
```

Kernel-Mode

```
th [r] [= StartAddress] [Count]
```

Parameters

Thread
Specifies threads to continue executing. All other threads are frozen. For more information about the syntax, see Thread Syntax. You can specify threads only in user mode.

r
Turns on and off the display of registers and flags. By default, the registers and flags are displayed. You can disable register display by using the **thr**, **pr**, **tr**, or .prompt_allow -reg commands. All of these commands control the same setting and you can use any of them to override any previous use of these commands.

You can also disable register display by using the l-os command. This setting is separate from the other four commands. To control which registers and flags are displayed, use the **rm (Register Mask)** command.

StartAddress
Specifies the address where the debugger begins execution. If you do not use *StartAddress*, execution begins at the instruction that the instruction pointer points to. For more information about the syntax, see Address and Address Range Syntax.

Count
Specifies the number of branching instructions that the debugger must encounter for the **th** command to end. The default value is one.

Environment

Modes	User mode, kernel mode
Targets	Live debugging only
Platforms	All

Additional Information

For more information about related commands, see Controlling the Target.

Remarks

The **th** command causes the target to begin executing. Execution continues until the debugger reaches a branching instruction or encounters a breakpoint.

If the program counter is already on a branching instruction, the debugger traces into the branching instruction and continues executing until another branching instruction is reached. This tracing, rather than execution, of the call is the only difference between **th** and **ph (Step to Next Branching Instruction)**.

th is available for all live sessions. This availability is the primary difference between **th** and **tb (Trace to Next Branch)**.

In source mode, you can associate one source line with multiple assembly instructions. This command does not stop at a branching instruction that is associated with the current source line.

tt (Trace to Next Return)

12/5/2018 • 2 minutes to read • Edit Online

The **tt** command executes the program until a return instruction is reached.

User-Mode

```
[~Thread] tt [r] [= StartAddress] [Count]
```

Kernel-Mode

```
tt [r] [= StartAddress] [Count]
```

Parameters

Thread
Specifies threads to continue executing. All other threads are frozen. For more information about the syntax, see Thread Syntax. You can specify threads only in user mode.

r
Turns on and off the display of registers and flags. By default, the registers and flags are displayed. You can disable register display by using the **ttr**, **pr**, **tr**, or .prompt_allow -reg commands. All of these commands control the same setting and you can use any of them to override any previous use of these commands.

You can also disable register display by using the l-os command. This setting is separate from the other four commands. To control which registers and flags are displayed, use the **rm (Register Mask)** command.

StartAddress
Specifies the address where the debugger begins execution. If you do not use *StartAddress*, execution begins at the instruction that the instruction pointer points to. For more information about the syntax, see Address and Address Range Syntax.

Count
Specifies the number of **return** instructions that the debugger must encounter for the **th** command to end. The default value is one.

Environment

Modes	User mode, kernel mode
Targets	Live debugging only
Platforms	All

Additional Information

For more information about related commands, see Controlling the Target.

Remarks

The **tt** command causes the target to begin executing. This execution continues until the debugger reaches a **return** instruction or encounters a breakpoint

If the program counter is already on a **return** instruction, the debugger traces into the return and continues executing until another **return** is reached. This tracing, rather than execution, of the call is the only difference between **tt** and **pt (Step to Next Return)**.

In source mode, you can associate one source line with multiple assembly instructions. This command does not stop at a **return** instruction that is associated with the current source line.

u, ub, uu (Unassemble)

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

The **u*** commands display an assembly translation of the specified program code in memory.

This command should not be confused with the **~u (Unfreeze Thread)** command.

```
u[u|b] Range
u[u|b] Address
u[u|b]
```

Parameters

Range
Specifies the memory range that contains the instructions to disassemble. For more information about the syntax, see Address and Address Range Syntax. If you use the **b** flag, you must specify *Range* by using the "*Address* **L***Length*" syntax, not the "*Address1 Address2*" syntax.

Address
Specifies the beginning of the memory range to disassemble. Eight instructions (on an x86-based processor) or nine instructions (on an Itanium-based processor) are unassembled. For more information about the syntax, see Address and Address Range Syntax.

b
Determines the memory range to disassemble by counting backward. If **ub** *Address* is used, the disassembled range will be the eight or nine byte range ending with *Address*. If a range is specified using the syntax **ub** *Address* **L***Length*, the disassembled range will be the range of the specified length ending at *Address*.

u
Specifies that the disassembly will continue even if there is a memory read error.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about assembly debugging and related commands, see Debugging in Assembly Mode.

Remarks

If you do not specify a parameter for the **u** command, the disassembly begins at the current address and extends eight instructions (on an x86-based or x64-based processor) or nine instructions (on an Itanium-based processor). When you use **ub** without a parameter, the disassembly includes the eight or nine instructions before the current address.

Do not confuse this command with the **up (Unassemble from Physical Memory)**. The **u** command disassembles only virtual memory, while the **up** command disassembles only physical memory.

uf (Unassemble Function)

12/5/2018 • 2 minutes to read • Edit Online

The **uf** command displays an assembly translation of the specified function in memory.

```
uf [Options] Address
```

Parameters

Options
One or more of the following options:

/c
Displays only the call instructions in a routine instead of the full disassembly. Call instructions can be useful for determination of caller and callee relationships from disassembled code.

/D
Creates linked callee names for navigation of the call graph.

/m
Relaxes the blocking requirements to permit multiple exits.

/o
Sorts the display by address instead of by function offset. This option presents a memory-layout view of a full function.

/O
Creates linked call lines for accessing call information and creating breakpoints.

/i
Displays the number of instructions in a routine.

Address
Specifies the address of the function to disassemble. For more information about the syntax, see Address and Address Range Syntax.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about assembly debugging and related commands, see Debugging in Assembly Mode.

Remarks

The display shows the whole function, according to the function order.

up (Unassemble from Physical Memory)

12/5/2018 • 2 minutes to read • Edit Online

The **up** command displays an assembly translation of the specified program code in physical memory.

```
up Range
up Address
up
```

Parameters

Range
Specifies the memory range in physical memory that contains the instructions to disassemble. For more information about the syntax, see Address and Address Range Syntax.

Address
Specifies the beginning of the memory range in physical memory to disassemble. Eight instructions (on an x86-based processor) or nine instructions (on an Itanium-based processor) are unassembled. For more information about the syntax, see Address and Address Range Syntax.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about assembly debugging and related commands, see Debugging in Assembly Mode.

Remarks

If you do not specify a parameter for the **up** command, the disassembly begins at the current address and extends eight instructions (on an x86-based processor) or nine instructions (on an Itanium-based processor).

Do not confuse this command with the **u (Unassemble)**. The **up** command disassembles only physical memory, while the **u** command disassembles only virtual memory.

ur (Unassemble Real Mode BIOS)

12/5/2018 • 2 minutes to read • Edit Online

The **ur** command displays an assembly translation of the specified 16-bit real-mode code.

```
ur Range
ur Address
ur
```

Parameters

Range
Specifies the memory range that contains the instructions to disassemble. For more information about the syntax, see Address and Address Range Syntax.

Address
Specifies the beginning of the memory range to disassemble. Eight instructions (on an x86-based processor) or nine instructions (on an Itanium-based processor) are unassembled. For more information about the syntax, see Address and Address Range Syntax.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about how to debug BIOS code, see Debugging BIOS Code.

Remarks

If you do not specify *Range* or *Address*, the disassembly begins at the current address and extends eight instructions (on an x86-based processor) or nine instructions (on an Itanium-based processor).

If you are examining 16-bit real-mode code on an x86-based processor, both the **ur** command and the **u (Unassemble)** command give correct results.

However, if real-mode code exists in a location where the debugger is not expecting it (for example, a non-x86 computer that is running or emulating x86-based BIOS code from a plug-in card), you must use **ur** to correctly disassemble this code.

If you use **ur** on 32-bit or 64-bit code, the command tries to disassemble the code as if it were 16-bit code. This situation produces meaningless results.

ux (Unassemble x86 BIOS)

12/5/2018 • 2 minutes to read • Edit Online

The **ux** command displays the instruction set of the x86-based BIOS code.

```
ux [Address]
```

Parameters

Address
Specifies the memory offset within the x86-based BIOS code. If you omit this parameter or specify zero, the default offset is the beginning of the BIOS.

Environment

Modes	Kernel mode only
Targets	Live debugging only
Platforms	x86-based only

Additional Information

For more information about how to debug BIOS code, see Debugging BIOS Code.

Remarks

The debugger displays the instructions that are generated from the first eight lines of code, beginning at the *Address* offset.

To make the **ux** command work correctly, HAL symbols must be available to the debugger. If the debugger cannot find these symbols, the debugger displays a "couldn't resolve" error.

vercommand (Show Debugger Command Line)

12/5/2018 • 2 minutes to read • Edit Online

The **vercommand** command displays the command that opened the debugger.

```
vercommand
```

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

version (Show Debugger Version)

12/5/2018 • 2 minutes to read • Edit Online

The **version** command displays version information about the debugger and all loaded extension DLLs. This command also displays the current version of the operating system of the target computer.

Do not confuse this command with the **!version (Show DLL Version)** extension command.

```
version
```

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

See also

CTRL+W (Show Debugger Version)

vertarget (Show Target Computer Version)

vertarget (Show Target Computer Version)

12/5/2018 • 2 minutes to read • Edit Online

The **vertarget** command displays the current version of the Microsoft Windows operating system of the target computer.

```
vertarget
```

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

See also

CTRL+W (Show Debugger Version)

version (Show Debugger Version)

wrmsr (Write MSR)

12/5/2018 • 2 minutes to read • Edit Online

The **wrmsr** command writes a value to a Model-Specific Register (MSR) at the specified address.

```
wrmsr Address Value
```

Parameters

Address
Specifies the address of the MSR.

Value
Specifies the 64-bit hexadecimal value to write to the MSR.

Environment

Modes	Kernel mode only
Targets	Live debugging only
Platforms	All

Remarks

The **wrmsr** command can display MSR's on x86-based, Itanium-based, and x64-based platforms. The MSR definitions are platform-specific.

See also

rdmsr (Read MSR)

wt (Trace and Watch Data)

12/5/2018 • 5 minutes to read • Edit Online

The **wt** command runs through the whole function and then displays statistics, when you execute this command at the beginning of a function call.

```
wt [WatchOptions] [= StartAddress] [EndAddress]
```

Parameters

WatchOptions
Specifies how to modify the display. You can use any of the following options.

OPTION	EFFECT
-l *Depth*	(User mode only) Specifies the maximum depth of the calls to display. Any calls that are at least *Depth* levels deeper than the starting point are executed silently.
-m *Module*	(User mode only) Restricts the display to code inside the specified module, plus the first level of calls made from that module. You can include multiple -m options to display code from multiple modules and no other modules.
-i *Module*	(User mode only) Ignores any code within the specified module. You can include multiple -i options to ignore code from multiple modules. If you use a -m option, the debugger ignores all -i options.
-ni	(User mode only) Does not display any entry into code that is being ignored because of an -m or -i option.
-nc	Does not display individual call information.
-ns	Does not display summary information.
-nw	Does not display warnings during the trace.
-oa	(User mode only) Displays the actual address of call sites.
-or	(User mode only) Displays the return register values of the called function, using the default radix as the base.

OPTION	EFFECT
-oR	(User mode only) Displays the return register values of the called function, in the appropriate type for each return value.

StartAddress
Specifies the address where the debugger begins execution. If you do not use *StartAddress*, execution begins at the instruction that the instruction pointer points to. For more information about the syntax, see Address and Address Range Syntax.

EndAddress
Specifies the address where tracing ends. If you do not use *EndAddress*, a single instruction or function call is executed.

Environment

Modes	User mode, kernel mode
Targets	Live debugging only
Platforms	**User mode:** all **Kernel mode:** x86-based only

Additional Information

For more information about issuing the **wt** command and an overview of related commands, see Controlling the Target.

Remarks

The **wt** command is useful if you want information about the behavior of a specific function, but you do not want to step through the function. Instead, go to the beginning of that function and then issue the **wt** command.

If the program counter is at a point that corresponds to a symbol (such as the beginning of a function or entry point into a module), the **wt** command traces until it reaches the current return address. If the program counter is on a **call** instruction, the **wt** command traces until it returns to the current location. This tracing is profiled in the Debugger Command window together with output that describes the various calls that the command encounters.

If the **wt** command is issued somewhere other than the beginning of a function, the command behaves like the **p (Step)** command. However, if you specify *EndAddress*, execution continues until that address is reached, even if this execution involves many program steps and function calls.

When you are debugging in source mode, you should trace into the function only to the point where you see the opening bracket of the function body. Then, you can use the **wt** command. (It is typically easier to insert a breakpoint at the first line of the function, or use Debug | Run to Cursor, and then use the **wt** command.)

Because the output from **wt** can be long, you might want to use a log file to record your output.

The following example shows a typical log file.

```
0:000> l+                 Source options set to show source lines
Source options are f:
     1/t - Step/trace by source line
     2/l - List source line for LN and prompt
     4/s - List source code at prompt
     8/o - Only show source code at prompt
0:000> p                  Not yet at the function call: use "p"
>  44:       minorVariableOne = 12;
0:000> p
>  45:       variableOne = myFunction(2, minorVariable);
0:000> t                  At the function call: now use "t"
MyModule!ILT+10(_myFunction):
0040100f e9cce60000      jmp     MyModule!myFunction (0040f6e0)
0:000> t
>  231:    {
0:000> wt                 At the function beginning:  now use "wt"
Tracing MyModule!myFunction to return address 00401137

  105     0 [  0] MyModule!myFunction
    1     0 [  1]   MyModule!ILT+1555(_printf)
    9     0 [  1]   MyModule!printf
    1     0 [  2]     MyModule!ILT+370(__stbuf)
   11     0 [  2]     MyModule!_stbuf
    1     0 [  3]       MyModule!ILT+1440(__isatty)
   14     0 [  3]       MyModule!_isatty
   50    15 [  2]     MyModule!_stbuf
   17    66 [  1]   MyModule!printf
    1     0 [  2]     MyModule!ILT+980(__output)
   59     0 [  2]     MyModule!_output
   39     0 [  3]       MyModule!write_char
  111    39 [  2]     MyModule!_output
   39     0 [  3]       MyModule!write_char

....

   11     0 [  5]           kernel32!__SEH_epilog4
   54 11675 [  4]         kernel32!ReadFile
  165 11729 [  3]       MyModule!_read
  100 11895 [  2]     MyModule!_filbuf
   91 11996 [  1]   MyModule!fgets
54545 83789 [  0] MyModule!myFunction
    1     0 [  1]   MyModule!ILT+1265(__RTC_CheckEsp)
    2     0 [  1]   MyModule!_RTC_CheckEsp
54547 83782 [  0] MyModule!myFunction

112379 instructions were executed in 112378 events (0 from other threads)

Function Name                               Invocations MinInst MaxInst AvgInst
MyModule!ILT+1265(__RTC_CheckEsp)                     1       1       1       1
MyModule!ILT+1440(__isatty)                          21       1       1       1
MyModule!ILT+1540(__ftbuf)                           21       1       1       1
....
ntdll!memcpy                                         24       1      40      19
ntdll!memset                                          2      29      29      29

23 system calls were executed

Calls  System Call
   23  ntdll!KiFastSystemCall
```

In the listing of the trace, the first number specifies the number of instructions that were executed, the second number specifies the number of instructions executed by child processes of the function, and the third number (in brackets) specifies the depth of the function in the stack (taking the initial function as zero). The indentation of the function name shows the call depth.

In the preceding example, **MyModule!myFunction** executes 105 instructions before it calls several subroutines, including **printf** and **fgets**, and then executes 54545 additional instructions after calling those functions, but before issuing a few more calls. However, in the final count, the display shows that **myFunction** executes 112,379 instructions, because this count includes all of the instructions that **myFunction** and its children execute. (The *children* of **myFunction** are functions that are called from **myFunction**, either directly or indirectly.)

In the preceding example, note also that **ILT+1440 (__isatty)** is called 21 times. In the final count, the summary of this function's behavior shows the number of times that it was called, the smallest number of instructions in any single execution, the largest number of instructions in any single execution, and the average number of instructions per execution.

If any system calls are made, they appear in the counter and are listed again at the end of the command output.

x (Examine Symbols)

8/8/2019 • 6 minutes to read • Edit Online

The **x** command displays the symbols in all contexts that match the specified pattern.

```
x [Options] Module!Symbol
x [Options] *
```

Parameters

Options
Specifies symbol searching options. You can use one or more of the following options:

/0
Displays only the address of each symbol.

/1
Displays only the name of each symbol.

/2
Displays only the address and name of each symbol (not the data type).

/D
Displays the output using Debugger Markup Language.

/t
Displays the data type of each symbol, if the data type is known.

/v
Displays the symbol type (local, global, parameter, function, or unknown) of each symbol. This option also displays the size of each symbol. The size of a function symbol is the size of the function in memory. The size of other symbols is the size of the data type that the symbol represents. Size is always measured in bytes and displayed in hexadecimal format.

/s *Size*
Display only those symbols whose size, in bytes, equals the value of *Size*. The *Size* of a function symbol is the size of the function in memory. The *Size* of other symbols is the size of the data type that the symbol represents. Symbols whose size cannot be determined are always displayed. *Size* must be a nonzero integer.

/q
Displays symbol names in quoted format.

/p
Omits the space before the opening parenthesis when the debugger displays a function name and its arguments. This kind of display can make it easier if you are copying function names and arguments from the **x** display to another location.

/f
Displays the data size of a function.

/d
Displays the data size of data.

/a
Sorts the display by address, in ascending order.

/A
Sorts the display by address, in descending order.

/n
Sorts the display by name, in ascending order.

/N
Sorts the display by name, in descending order.

/z
Sorts the display by size, in ascending order.

/Z
Sorts the display by size, in descending order.

Module
Specifies the module to search. This module can be an .exe, .dll, or .sys file. *Module* can contain a variety of wildcard characters and specifiers. For more information about the syntax, see String Wildcard Syntax.

Symbol
Specifies a pattern that the symbol must contain. *Symbol* can contain a variety of wildcard characters and specifiers. For more information about the syntax, see String Wildcard Syntax.

Because this pattern is matched to a symbol, the match is not case sensitive, and a single leading underscore (_) represents any quantity of leading underscores. You can add spaces within *Symbol*, so that you can specify symbol names that contain spaces (such as "operator new" or "Template<A, B>") without using wildcard characters.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

The following command finds all of the symbols in MyModule that contain the string "spin".

```
0:000> x mymodule!*spin*
```

The following command quickly locates the "DownloadMinor" and "DownloadMajor" symbols in MyModule.

```
0:000> x mymodule!downloadm??or
```

You can also show all symbols in the MyModule by using the following command.

```
0:000> x mymodule!*
```

The preceding commands also force the debugger to reload symbol information from MyModule. If you want to reload the symbols in the module with a minimal display, use the following command.

```
0:000> x mymodule!*start*
```

A few symbols always contain the string "start". Therefore, the preceding command always displays some output to verify that the command works. But the preceding command avoids the excessive display length of **x mymodule!***.

The display shows the starting address of each symbol and the full symbol name. If the symbol is a function name, the display also includes a list of its argument types. If the symbol is a global variable, its current value is displayed.

There is one other special case of the **x** command. To display the addresses and names of all local variables for the current context, use the following command.

```
0:000> x *
```

Note In most cases, you cannot access local variables unless private symbols have been loaded. For more information about this situation, see dbgerr005: Private Symbols Required. To display the values of local variables, use the **dv (Display Local Variables)** command.

The following example illustrates the **/0**, **/1**, and **/2** options.

```
0:000:x86> x /0 MyApp!Add*
00b51410
00b513d0

0:000:x86> x /1 MyApp!Add*
MyApp!AddThreeIntegers
MyApp!AddTwoIntegers

0:000:x86> x /2 MyApp!Add*
00b51410          MyApp!AddThreeIntegers
00b513d0          MyApp!AddTwoIntegers
```

The **/0**, **/1**, and **/2** options are useful if you want to use the output of the **x** command as input to the **.foreach** command.

```
.foreach ( place { x /0 MyApp!*MySym*} ) { .echo ${place}+0x18 }
```

The following example demonstrates the switch **/f** when used to filter functions on the module notepad.exe.

```
0:000> x /f /v notepad!*main*
prv func   00000001`00003340  249 notepad!WinMain (struct HINSTANCE__ *, struct HINSTANCE__ *, char *, int)
prv func   00000001`0000a7b0   1c notepad!WinMainCRTStartup$filt$0 (void)
prv func   00000001`0000a540  268 notepad!WinMainCRTStartup (void)
```

When you use the **/v** option, the first column of the display shows the symbol type (local, global, parameter, function, or unknown). The second column is the address of the symbol. The third column is the size of the symbol, in bytes. The fourth column shows the module name and symbol name. In some cases, this display is followed by an equal sign (=) and then the data type of the symbol. The source of the symbol (public or full symbol information) is also displayed.

```
kd> x /v nt!CmType*
global 806c9e68    0 nt!CmTypeName = struct _UNICODE_STRING []
global 806c9e68  150 nt!CmTypeName = struct _UNICODE_STRING [42]
global 806c9e68    0 nt!CmTypeName = struct _UNICODE_STRING []
global 805bd7b0    0 nt!CmTypeString = unsigned short *[]
global 805bd7b0   a8 nt!CmTypeString = unsigned short *[42]
```

In the preceding example, the size is given in hexadecimal format, while the data type is given in decimal format. Therefore, in the last line of the preceding example, the data type is an array of 42 pointers to unsigned short integers. The size of this array is 42*4 = 168, and 168 is displayed in hexadecimal format as 0xA8.

You can use the **/s***Size* option to display only those symbols whose size, in bytes, is a certain value. For example, you can restrict the command in the preceding example to symbols that represent objects whose size is 0xA8.

```
kd> x /v /s a8 nt!CmType*
global 805bd7b0   a8 nt!CmTypeString = unsigned short *[42]
```

Working With Data Types

The **/t** option causes the debugger to display information about each symbol's data type. Note that for many symbols, this information is displayed even without the **/t** option. When you use **/t**, such symbols have their data type information displayed twice.

```
0:001> x prymes!__n*
00427d84 myModule!__nullstring = 0x00425de8 "(null)"
0042a3c0 myModule!_nstream = 512
Type information missing error for _nh_malloc
004021c1 myModule!MyStructInstance = struct MyStruct
00427d14 myModule!_NLG_Destination = <no type information>

0:001> x /t prymes!__n*
00427d84 char * myModule!__nullstring = 0x00425de8 "(null)"
0042a3c0 int myModule!_nstream = 512
Type information missing error for _nh_malloc
004021c1 struct MyStruct myModule!MyStructInstance = struct MyStruct
00427d14 <NoType> myModule!_NLG_Destination = <no type information>
```

The x command will display an instance of a type.

```
0:001> x foo!MyClassInstance
00f4f354          foo!MyClassInstance = 0x00f78768
```

The x command does not display anything based on just the name of a type.

```
0:001> x foo!MyClass
0:001>
```

To display type information using the name of a type, consider using **dt (Display Type)**, it provides information for both types and instances of types:

```
0:001> dt foo!MyClass
   +0x000 IntMemberVariable : Int4B
   +0x004 FloatMemberVariable : Float
   +0x008 BoolMemberVariable : Bool
   +0x00c PtrMemberVariable : Ptr32 MyClass
```

Working With Templates

You can use wild cards with the x command to display template classes as shown in this sample.

```
0:001>  x Fabric!Common::ConfigEntry*TimeSpan?
000007f6`466a2f9c Fabric!Common::ConfigEntry<Common::TimeSpan>::ConfigEntry<Common::TimeSpan> (void)
000007f6`466a3020 Fabric!Common::ConfigEntry<Common::TimeSpan>::~ConfigEntry<Common::TimeSpan> (void)
```

Consider using the **dt (Display Type)** command when working with templates, as the x command does not display individual template class items.

```
0:001> dt foo!Common::ConfigEntry<Common::TimeSpan>
   +0x000 __VFN_table : Ptr64
   +0x008 componentConfig_ : Ptr64 Common::ComponentConfig
   +0x010 section_         : std::basic_string<wchar_t,std::char_traits<wchar_t>,std::allocator<wchar_t> >
   +0x038 key_             : std::basic_string<wchar_t,std::char_traits<wchar_t>,std::allocator<wchar_t> >
```

See also

Verifying Symbols

dv (Display Local Variables)

z (Execute While)

12/5/2018 • 2 minutes to read • Edit Online

The **z** command executes a command while a given condition is true.

User-Mode

```
Command ; z( Expression )
```

Kernel-Mode

```
Command ; [Processor] z( Expression )
```

Parameters

Command
Specifies the command to execute while the *Expression* condition evaluates to a nonzero value. This command is always executed at least once.

Processor
Specifies the processor that applies to the test. For more information about the syntax, see Multiprocessor Syntax. You can specify processors only in kernel mode.

Expression
Specifies the condition to test. If this condition evaluates to a nonzero value, the *Command* command is executed again and then *Expression* is tested again. For more information about the syntax, see Numerical Expression Syntax.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

In many debugger commands, the semicolon is used to separate unrelated commands. However, in the **z** command, a semicolon separates the "z" from the *Command* parameter.

The *Command* command is always executed at least once, and then *Expression* is tested. If the condition is a nonzero value, the command is again executed, and then *Expression* is tested again. (This behavior is similar to a C-language **do - while** loop, not a simple **while** loop.)

If there are several semicolons to the left of the "z", all commands to the left of the "z" repeat as long as the *Expression* condition is true. Such commands can be any debugger commands that permit a terminal semicolon.

If you add another semicolon and additional commands after the **z** command, these additional commands are executed after the loop is complete. We do not typically recommend a line that begins with "z" because it generates uninteresting output forever unless the condition becomes false because of some other action. Note that you can nest **z** commands.

To break a loop that is continuing for too long, use **CTRL+C** in CDB or KD, or use Debug | Break or CTRL+BREAK in WinDbg.

The following code example shows an unnecessarily complex way to zero the **eax** register.

```
0:000> reax = eax - 1 ; z(eax)
```

The following example increments the **eax** and **ebx** registers until one of them is at least 8 and then it increments the **ecx** register once.

```
0:000> reax=eax+1; rebx=ebx+1; z((eax<8)|(ebx<8)); recx=ecx+1
```

The following example uses C++ expression syntax and uses the pseudo-register **$t0** as a loop variable.

```
0:000> .expr /s c++
Current expression evaluator: C++ - C++ source expressions

0:000> db pindexcreate[@$t0].szKey; r$t0=@t0+1; z( @$t0 < cIndexCreate )
```

See also

j (Execute If-Else)

Meta-Commands

12/5/2018 • 2 minutes to read • Edit Online

This section of the reference discusses the various debugger *meta-commands* that can be used in CDB, KD, and WinDbg. These commands are preceded by a period (.).

.abandon (Abandon Process)

12/5/2018 • 2 minutes to read • Edit Online

The **.abandon** command ends the debugging session, but leaves the target application in a debugging state. This returns the debugger to dormant mode.

```
.abandon [/h|/n]
```

Parameters

/h
Any outstanding debug event will be continued and marked as handled. This is the default.

/n
Any outstanding debug event will be continued unhandled.

Environment

This command is only supported in Windows XP and later versions of Windows.

Modes	user mode only
Targets	live debugging only
Platforms	all

Additional Information

If the target is left in a debugging state, a new debugger can be attached to it. See Re-attaching to the Target Application for details. However, after a process has been abandoned once, it can never be restored to a running state without a debugger attached.

.allow_exec_cmds (Allow Execution Commands)

12/5/2018 • 2 minutes to read • Edit Online

The **.allow_exec_cmds** command controls whether execution commands can be used.

```
.allow_exec_cmds 0
.allow_exec_cmds 1
.allow_exec_cmds
```

Parameters

0

Prevents execution commands from being used.

1

Allows execution commands to be used.

Environment

Modes	user mode and kernel mode
Targets	live debugging only
Platforms	all

Additional Information

For a complete list of execution commands, see Controlling the Target.

Remarks

With no parameters, **.allow_exec_cmds** will display whether execution commands are currently permitted.

Execution commands include **g (Go)**, **t (Trace)**, **p (Step)**, and any other command or WinDbg graphical interface action that would cause the target to execute.

.allow_image_mapping (Allow Image Mapping)

12/5/2018 • 2 minutes to read • Edit Online

The **.allow_image_mapping** command controls whether image files will be mapped.

```
.allow_image_mapping [/r] 0
.allow_image_mapping [/r] 1
.allow_image_mapping
```

Parameters

/r
Reloads all modules in the debugger's module list. This is equivalent to **.reload /d**.

0
Prevents image files from being mapped.

1
Allows image files to be mapped.

Environment

Modes	user mode and kernel mode
Targets	live, crash dump
Platforms	all

Remarks

With no parameters, **.allow_image_mapping** will display whether image file mapping is currently allowed. By default, this mapping is allowed.

Image mapping is most common when a minidump is being debugged. Image mapping can also occur if DbgHelp is unable to access debug records (for example, during kernel debugging when memory has been paged out).

.apply_dbp (Apply Data Breakpoint to Context)

1/11/2019 • 2 minutes to read • Edit Online

The **.apply_dbp** command applies the current process' existing data breakpoints to the specified register context.

```
.apply_dbp [/m Context]
```

Parameters

/m *Context*
Specifies the address of a register context (CONTEXT structure) in memory to which to apply the current process' data breakpoints.

Environment

Modes	user mode and kernel mode
Targets	live target only
Platforms	all

Additional Information

For more information about breakpoints controlled by the processor, see Processor Breakpoints (ba Breakpoints). For more information about the register context (thread context), see Register Context.

Remarks

Breakpoints that are controlled by the processor are called *data breakpoints* or *processor breakpoints*. These breakpoints are created by the **ba (Break on Access)** command.

These breakpoints are associated with a memory location in the address space of a specific process. The **.apply_dbp** command modifies the specified register context so that these data breakpoints will be active when this context is used.

If the **/m** *Address* parameter is not used, data breakpoints will be applied to the current register context.

This command can only be used if the target is in native machine mode. For example, if the target is running on a 64-bit machine emulating an x86 processor using *WOW64*, this command cannot be used.

One example of a time this command is useful is when you are in an exception filter. The **.apply_dbp** command can update the exception filter's stored context. Data breakpoints will then be applied when the exception filter exits and the stored context is resumed. Without such a modification it is possible that data breakpoints would be lost.

.asm (Change Disassembly Options)

12/5/2018 • 2 minutes to read • Edit Online

The **.asm** command controls how disassembly code will be displayed.

```
.asm
.asm[-] Options
```

Parameters

-

Causes the specified options to be disabled. If no minus sign is used, the specified options will be enabled.

Options

Can be any number of the following options:

ignore_output_width

Prevents the debugger from checking the width of lines in the disassembly display.

no_code_bytes

(x86 and x64 targets only) Suppresses the display of raw bytes.

source_line

Prefixes each line of disassembly with the line number of the source code.

verbose

(Itanium target only) Causes bundle-type information to be displayed along with the standard disassembly information.

Environment

Modes	user mode, kernel mode
Targets	live, crash dump
Platforms	all

Additional Information

For a description of assembly debugging and related commands, see Debugging in Assembly Mode.

Remarks

Using **.asm** by itself displays the current state of the options.

This command affects the display of any disassembly instructions in the Debugger Command window. In WinDbg it also changes the contents of the Disassembly window.

.attach (Attach to Process)

1/3/2019 • 2 minutes to read • Edit Online

The **.attach** command attaches to a new target application.

```
.attach [-premote RemoteOptions] AttachOptions PID
```

Parameters

RemoteOptions
Specifies a process server to attach to. The options are the same as those for the command line **-premote** option. See **Activating a Smart Client** for syntax details.

AttachOptions
Specifies how the attach is to be done. This can include any of the following options:

-b
Prevents the debugger from requesting an initial break-in when attaching to a target process. This can be useful if the application is already suspended, or if you want to avoid creating a break-in thread in the target.

-e
Allows the debugger to attach to a process that is already being debugged. See Re-attaching to the Target Application for details.

-k
Begins a local kernel debugging session. See Performing Local Kernel Debugging for details.

-f
Freezes all threads in all target applications, except in the new target being attached to. These threads will remain frozen until an exception occurs in the newly-attached process. Note that an initial breakpoint qualifies as an exception. Individual threads can be unfrozen by using the **~u (Unfreeze Thread)** command.

-r

Causes the debugger to start the target process running when it attaches to it. This can be useful if the application is already suspended and you want it to resume execution.

-v
Causes the specified process to be debugged noninvasively.

PID
Specifies the process ID of the new target application.

Environment

Modes	user mode only
Targets	live debugging only
Platforms	all

Remarks

This command can be used when CDB is dormant, or if it is already debugging one or more processes. It cannot be used when WinDbg is dormant.

If this command is successful, the debugger will attach to the specified process the next time the debugger issues an execution command. If this command is used several times in a row, execution will have to be requested as many times as this command was used.

Because execution is not permitted during noninvasive debugging, the debugger is not able to noninvasively debug more than one process at a time. This also means that using the **.attach -v** command may render an already-existing invasive debugging session less useful.

Multiple target processes will always be executed together, unless some of their threads are frozen or suspended.

If you wish to attach to a new process and freeze all your existing targets, use the **-f** option. For example, you might be debugging a crash in a client application and want to attach to the server process without letting the client application continue running.

If the **-premote** option is used, the new process will be part of a new system. For details, see Debugging Multiple Targets.

.beep (Speaker Beep)

12/5/2018 • 2 minutes to read • Edit Online

The **.beep** command makes noise on the computer speaker.

```
.beep
```

Environment

This command cannot be used in script files.

Modes	user mode, kernel mode
Targets	live, crash dump
Platforms	all

.bpcmds (Display Breakpoint Commands)

12/5/2018 • 2 minutes to read • Edit Online

The **.bpcmds** command displays the commands that were used to set each of the current breakpoints.

```
.bpcmds
```

Environment

Modes	user mode, kernel mode
Targets	live, crash dump
Platforms	all

Additional Information

For more information about and examples of how to use breakpoints, other breakpoint commands and methods of controlling breakpoints, see Using Breakpoints.

Remarks

If it is unclear whether a particular breakpoint is set at an address, at a symbolic reference, or at a symbol, use the **.bpcmds** command to shows which breakpoint command was used to create it. The command that was used to create a breakpoint determines its nature:

- The **bp (Set Breakpoint)** command sets a breakpoint at an address.
- The **bu (Set Unresolved Breakpoint)** command sets a breakpoint on a symbolic reference.
- The **bm (Set Symbol Breakpoint)** command sets a breakpoint on symbols that match a specified pattern. If the **/d** switch is included, it creates zero or more breakpoints on addresses (like bp), otherwise it creates zero or more breakpoints on symbolic references (like bu).
- The **ba (Break on Access)** command sets a data breakpoint at an address.

The output of **.bpcmds** reflects the current nature of each breakpoint. Each line of the **.bpcmds** display begins with the command used to create it (**bp**, **bu**, or **ba**) followed by the breakpoint ID, and then the location of the breakpoint.

If the breakpoint was created by **ba**, the access type and size are displayed as well.

If the breakpoint was created by **bm** without the **/d** switch, the display indicates the breakpoint type as **bu**, followed by the evaluated symbol enclosed in the **@!""** token (which indicates it is a literal symbol and not a numeric expression or register). If the breakpoint was created by **bm** with the **/d** switch, the display indicates the breakpoint type as **bp**.

Here is an example:

```
0:000> bp notepad!winmain

0:000> .bpcmds
bp0 0x00000001`00003340 ;

0:000> bu myprog!winmain
breakpoint 0 redefined

0:000> .bpcmds
bu0 notepad!winmain;

0:000> bu myprog!LoadFile

0:000> bp myprog!LoadFile+10

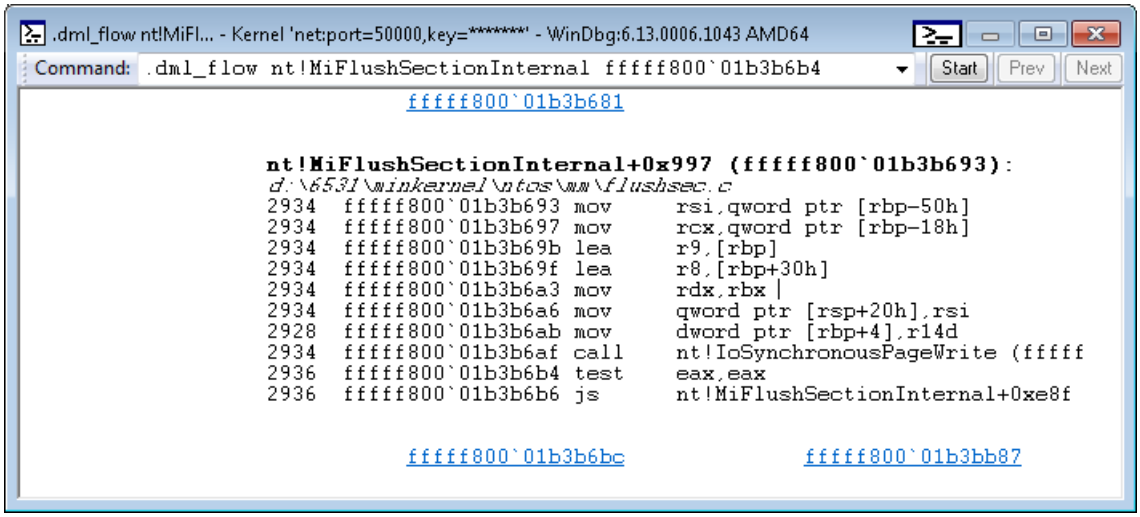
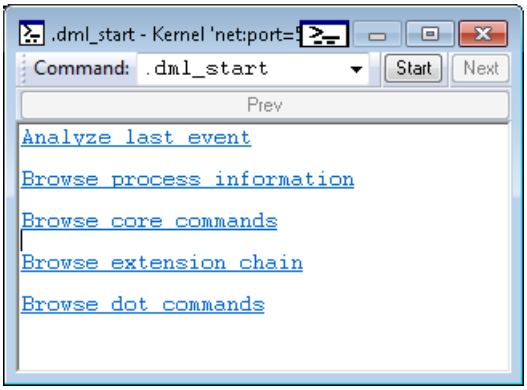
0:000> bm myprog!openf*
  3: 00421200 @!"myprog!openFile"
  4: 00427800 @!"myprog!openFilter"

0:000> bm /d myprog!closef*
  5: 00421600 @!"myprog!closeFile"

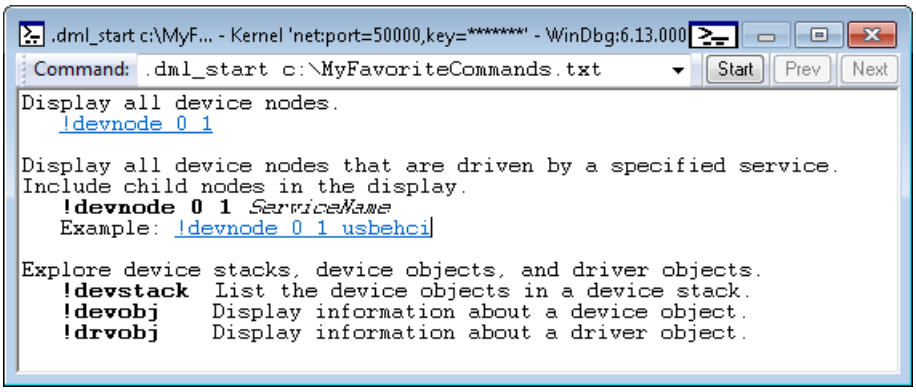
0:000> ba r2 myprog!LoadFile+2E

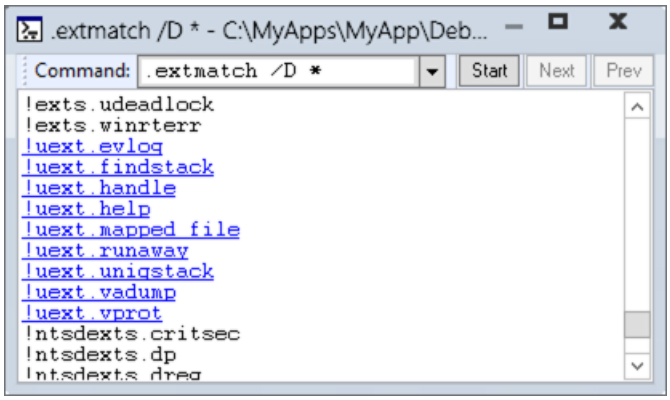
0:000> .bpcmds
bu0 notepad!winmain;
bu1 notepad!LoadFile;
bp2 0x0042cc10 ;
bu3 @!"myprog!openFile";
bu4 @!"myprog!openFilter";
bp5 0x00421600 ;
ba6 r2 0x0042cc2e ;
```

In this example, notice that the output of **.bpcmds** begins with the relevant command ("bu", "bp", or "ba"), followed by the breakpoint number (with no intervening space).

Notice that because breakpoint number 0 was originally set using **bp**, and then was redefined using **bu**, the display shows its type as "bu".

Notice also that breakpoints 3, 4, and 5, which were created by the **bm** commands shown in this example, are displayed as either type "bp" or type "bu", depending on whether the **/d** switch was included when **bm** was used.

.bpsync (Synchronize Threads at Breakpoint)

12/5/2018 • 2 minutes to read • Edit Online

When a thread reaches a breakpoint, the **.bpsync** command freezes all other threads, until the thread to which the breakpoint applies has stepped through the breakpoint.

```
.bpsync 1
.bpsync 0
.bpsync
```

Parameters

1
Causes all threads to freeze when one thread has reached a breakpoint. After the thread to which the breakpoint applies has stepped through the breakpoint, the other threads are unfrozen.

0
Allows other threads to continue executing when one thread has reached a breakpoint. This is the default behavior.

Environment

Modes	user mode only
Targets	live, crash dump
Platforms	all

Remarks

With no parameters. the **.bpsync** command displays the current rule governing breakpoint synchronization behavior.

The **.bpsync** command applies both to software breakpoints (set with **bp**, **bu**, or **bm**) and to processor breakpoints (set with **ba**).

If there is a possibility of multiple threads running through the same code, the **.bpsync 1** command can be useful for capturing all breakpoint occurrences. Without this command, a breakpoint occurrence could be missed because the first thread to reach the breakpoint always causes the debugger to temporarily remove the breakpoint. In the short period when the breakpoint is removed, other threads could reach the same place in the code and not trigger the breakpoint as intended.

The temporary removal of breakpoints is a normal aspect of debugger operation. When the target reaches a breakpoint and is resumed, the debugger has to remove the breakpoint temporarily so that the target can execute the real code. After the real instruction has been executed, the debugger reinserts the break. To do this, the debugger restores the code (or turns off data breaks), does a single-step, and then puts the break back.

Note that if you use **.bpsync 1**, there is a risk of deadlocks among the threads that have been frozen.

.breakin (Break to the Kernel Debugger)

12/5/2018 • 2 minutes to read • Edit Online

The **.breakin** command switches from user-mode debugging to kernel-mode debugging. This command is particularly useful when you are controlling the user-mode debugger from the kernel debugger.

```
.breakin
```

Environment

Modes	user mode only
Targets	live debugging only
Platforms	all

Remarks

If kernel-mode debugging was enabled during the boot process and you are running a user-mode debugger, you can use the **.breakin** command to halt the operating system and transfer control to a kernel debugger.

The **.breakin** command causes a kernel-mode break in the debugger's process context. If a kernel debugger is attached, it will become active. The kernel debugger's process context will automatically be set to the process of the user-mode debugger, not the user-mode debugger's target process.

This command is primarily useful when debugging a user-mode problem requires retrieving information about the kernel state of the system. Resuming execution in the kernel debugger is necessary before the user-mode debugging session can continue.

When you are controlling the user-mode debugger from the kernel debugger and the user-mode debugger prompt is visible in the kernel debugger, this command will pause the user-mode debugger and make the kernel-mode debugging prompt appear.

If the system is unable to break into the kernel debugger, an error message is displayed.

This command is also useful if you use the kernel debugger to set a breakpoint in user space and that breakpoint is caught by a user-mode debugger instead of the kernel debugger. Issuing this command in the user-mode debugger will transfer control to the kernel debugger.

If the **.breakin** command is used on a system that was not booted with debugging enabled, it has no effect.

.browse (Display Command in Browser)

12/5/2018 • 2 minutes to read • Edit Online

The **.browse** command displays the output of a specified command in a new Command Browser window.

```
.browse Command
```

Parameters

Command
The command to be executed and displayed in a new Command Browser window.

Remarks

The following example uses the **.browse** command to display the output of the **.chain /D** command in a Command Browser window.

```
.browse .chain /D
```

.bugcheck (Display Bug Check Data)

12/5/2018 • 2 minutes to read • Edit Online

The **.bugcheck** command displays the data from a bug check on the target computer.

```
.bugcheck
```

Environment

Modes	kernel mode only
Targets	live, crash dump
Platforms	all

Additional Information

For more information about bug checks, see Bug Checks (Blue Screens). For a description of individual bug checks, see the Bug Check Code Reference section.

Remarks

This command displays the current bug check data. (This bug check data will be accessible until the crashed machine is rebooted.)

You can also display bug check data on 32-bit systems by using **dd NT!KiBugCheckData L5**, or on 64-bit systems by using **dq NT!KiBugCheckData L5**. However, the .bugcheck command is more reliable, because it works in some scenarios that the **d* (Display Memory)** command will not (such as user-mode minidumps).

The **!analyze** extension command is also useful after a bug check occurs.

.cache (Set Cache Size)

12/5/2018 • 2 minutes to read • Edit Online

The **.cache** command sets the size of the cache used to hold data obtained from the target. Also sets a number of cache and memory options.

```
.cache Size
.cache Option
.cache
```

Parameters

Size
The size of the kernel debugging cache, in kilobytes. If *Size* is zero, the cache is disabled. The command output displays the cache size in bytes. (The default size is 1000 KB.)

Option
Can be any one of the following options:

hold
Automatic cache flushing is disabled.

unhold
Turns off the **hold** option. (This is the default setting.)

decodeptes
All transition page table entries (PTEs) will be implicitly decoded. (This is the default setting.)

nodecodeptes
Turns off the **decodeptes** option.

forcedecodeptes
All virtual addresses will be translated into physical addresses before access. This option also causes the cache to be disabled. Unless you are concerned with kernel-mode memory, it is more efficient to use **forcedecodeuser** instead.

forcedecodeuser
All user-mode virtual addresses will be translated into physical addresses before access. This option also causes the cache to be disabled.

Note You must activate **forcedecodeuser** (or **forcedecodeptes**) before using **.thread (Set Register Context)**, **.context (Set User-Mode Address Context)**, **.process (Set Process Context)**, or **!session** during live debugging. If you use the **/p** option with **.thread** and **.process**, the **forcedecodeuser** option is automatically set. In any other case, you will need to use the **.cache forcedecodeuser** command explicitly.

noforcedecodeptes
Turns off the **forcedecodeptes** and **forcedecodeuser** options. (This is the default setting.)

flushall
Deletes the entire virtual memory cache.

flushu
Deletes all entries of ranges with errors from the cache, as well as all user-mode entries.

flush *Address*
Deletes a 4096-byte block of the cache, beginning at *Address*.

Environment

Modes	kernel mode only
Targets	live debugging only
Platforms	all

Remarks

If **.cache** is used with no arguments, the current cache size, status, and options are displayed.

The **.cache forcedecodeuser** or **.cache forcedecodeptes** option will only last as long as the debugger remains broken into the target computer. If any stepping or execution of the target takes place, the **noforcedecodeptes** state will again take effect. This prevents the debugger from interfering with execution or a reboot in an unproductive manner.

.call (Call Function)

12/5/2018 • 4 minutes to read • Edit Online

The **.call** command causes the target process to execute a function.

```
.call [/v] Function( Arguments )
.call /s Prototype Function( Arguments )
.call /c
.call /C
```

Parameters

/v
Verbose information about the call and its arguments is displayed.

***/s** Prototype*
Allows you to call the function that is specified by *Function* even though you do not have the correct symbols. In this case, you must have symbols for another function that has the same calling prototype as the function you are trying to call. The *Prototype* parameter is the name of this prototype function.

Function
Specifies the function being called. This can be the name of the function (preferably qualified with a module name), or any other expression that evaluates to the function address. If you need to call a constructor or destructor, you must supply the address -- or else use a C++ expression to evaluate named syntax for the operators (see Numerical Expression Syntax for details).

Arguments
Specifies the arguments passed to the function. If you are calling a method, the first argument must be **this**, and all other arguments follow it. Arguments should be separated with commas and should match the usual argument syntax. Variable-length argument lists are supported. Expressions within an argument are parsed by the C++ expression evaluator; see C++ Numbers and Operators for details. You cannot enter a literal string as an argument, but you can use a pointer to a string, or any other memory accessible to the target process.

/c
Clears any existing call on the current thread.

/C
Clears any existing call on the current thread, and resets the context of the current thread to the context stored by the outstanding call.

Environment

Modes	user mode only
Targets	live debugging only
Platforms	x86 and x64 only

Remarks

The specified function is called by the current thread of the current process.

Only the **cdecl**, **stdcall**, **fastcall**, and **thiscall** calling conventions are supported. Managed code cannot be called by this command.

After **.call** is used, the debugger will update the stack, change the instruction pointer to point to the beginning of the called function, and then stop. Use **g (Go)** to resume execution, or **~. g** to execute just the thread making the call.

When the function returns, a break occurs and the debugger displays the return value of the function. The return value is also stored in the **$callret** pseudo-register, which acquires the type of the return value.

If you have broken into the target using CTRL+C or CTRL+BREAK, the current thread is an additional thread created to handle the breakin. If you issue a **.call** command at this point, the extra thread will be used for the called function.

If you have reached a predefined breakpoint, there is no extra breakin thread. If you use **.call** while at a breakpoint in user mode, you could use **g** to execute the entire process, or **~. g** to execute just the current thread. Using **g** may distort your program's behavior, since you have taken one thread and diverted it to this new function. On the other hand, this thread will still have its locks and other attributes, and thus **~. g** may risk deadlocks.

The safest way to use **.call** is to set a breakpoint in your code at a location where a certain function could be safely called. When that breakpoint is hit, you can use **.call** if you desire that function to run. If you use **.call** at a point where this function could not normally be called, a deadlock or target corruption could result.

It may be useful to add extra functions to your source code that are not called by the existing code, but are intended to be called by the debugger. For example, you could add functions that are used to investigate the current state of your code and its environment and store information about the state in a known memory location. Be sure not to optimize your code, or these functions may be removed by the compiler. Use this technique only as a last resort, because if your application crashes **.call** will not be available when debugging the dump file.

The **.call /c** and **.call /C** commands should only be used if an attempt to use **.call** has failed, or if you changed your mind before entering the **g** command. These should not be used casually, since abandoning an uncompleted call can lead to a corrupted target state.

The following code example shows how the **.call /s** command is used.

```
.call /s KnownFunction UnknownFunction( 1 )
```

In this example, you have private symbols for **KnownFunction**, which takes an integer as its only argument and returns, for example, a pointer to an array. You do not have symbols, or possibly you only have public symbols for **UnknownFunction**, but you do know that it takes an integer as its only argument and returns a pointer to an array. By using the **/s** option, you can specify that **UnknownFunction** will work the same way that **KnownFunction** does. Thus, you can successfully generate a call to **UnknownFunction**.

.chain (List Debugger Extensions)

12/5/2018 • 2 minutes to read • Edit Online

The **.chain** command lists all loaded debugger extensions in their default search order.

```
.chain
.chain /D
```

Parameters

/D

Displays the output using Debugger Markup Language. In the output, each listed module is a link that you can click to get information about the extensions that are implemented by the module.

Environment

Modes	user mode, kernel mode
Targets	live, crash dump
Platforms	all

Additional Information

For details on loading, unloading, and controlling extensions, see Loading Debugger Extension DLLs. For details on executing extension commands and an explanation of the default search order, see Using Debugger Extension Commands.

.childdbg (Debug Child Processes)

12/5/2018 • 2 minutes to read • Edit Online

The **.childdbg** command controls the debugging of child processes.

```
.childdbg 0
.childdbg 1
.childdbg
```

Parameters

0

Prevents the debugger from debugging child processes.

1

Causes the debugger to debug child processes.

Environment

This command is only supported in Windows XP and later versions of Windows.

Modes	user mode only
Targets	live debugging only
Platforms	x86, x64, and Itanium only

Remarks

Child processes are additional processes launched by the original target application.

With no parameters, **.childdbg** will display the current status of child-process debugging.

.clients (List Debugging Clients)

12/5/2018 • 2 minutes to read • Edit Online

The **.clients** command lists all debugging clients currently connected to the debugging session.

```
.clients
```

Environment

Modes	user mode, kernel mode
Targets	live, crash dump
Platforms	all

Additional Information

For more details and other commands that can be used while performing remote debugging through the debugger, see Controlling a Remote Debugging Session.

.closehandle (Close Handle)

12/5/2018 • 2 minutes to read • Edit Online

The **.closehandle** command closes a handle owned by the target application.

```
.closehandle Handle
.closehandle -a
```

Parameters

Handle
Specifies the handle to be closed.

-a
Causes all handles owned by the target application to be closed.

Environment

Modes	user mode only
Targets	live debugging only
Platforms	all

Remarks

You can use the **!handle** extension to display the existing handles.

.cls (Clear Screen)

12/5/2018 • 2 minutes to read • Edit Online

The **.cls** command clears the Debugger Command window display.

```
.cls
```

Environment

Modes	user mode, kernel mode
Targets	live, crash dump
Platforms	all

.context (Set User-Mode Address Context)

12/5/2018 • 2 minutes to read • Edit Online

The **.context** command specifies which page directory of a process will be used for the user-mode address context, or displays the current user-mode address context.

```
.context [PageDirectoryBase]
```

Parameters

PageDirectoryBase
Specifies the base address for a page directory of a desired process. The user-mode address context will be set to this page directory. If *PageDirectoryBase* is zero, the user-mode address context will be set to the page directory for the current system state. If *PageDirectoryBase* is omitted, the current user-mode address context is displayed.

Environment

Modes	kernel mode only
Targets	live, crash dump
Platforms	all

Additional Information

For more information about the user-mode address context and other context settings, see Changing Contexts.

Remarks

Generally, when you are doing kernel debugging, the only visible user-mode address space is the one associated with the current process.

The **.context** command instructs the kernel debugger to use the specified page directory as the *user-mode address context*. After this command is executed, the debugger will have access to this virtual address space. The page tables for this address space will be used to interpret all user-mode memory addresses. This allows you to read and write to this memory.

The **.process (Set Process Context)** command has a similar effect. However, the **.context** command sets the user-mode address context to a specific page directory, while the **.process** command sets the *process context* to a specific process. On an x86 processor, these two commands have essentially the same effect. However, on an Itanium processor, a single process may have more than one page directory. In this case, the **.process** command is more powerful, because it will allow access to all the page directories associated with a process. See Process Context for more details.

If you are doing live debugging, you should issue a **.cache forcedecodeuser** command in addition to the **.context** command. This forces the debugger to look up the physical addresses of the memory space needed. (This can be slow, because it often means a huge amount of data must be transferred across the debug cable.)

If you are doing crash dump debugging, the **.cache** command is not needed. However, you will not have access to

any portions of the virtual address space of the user-mode process that were paged to disk when the crash occurred.

Here is an example. Use the **!process** extension to find the directory base for the desired process:

```
kd> !process 0 0
**** NT ACTIVE PROCESS DUMP ****
PROCESS fe5039e0  SessionId: 0  Cid: 0008    Peb: 00000000  ParentCid: 0000
    DirBase: 00030000  ObjectTable: fe529b68  TableSize:  50.
    Image: System

...

PROCESS fe3c0d60  SessionId: 0  Cid: 0208    Peb: 7ffdf000  ParentCid: 00d4
 DirBase: 0011f000  ObjectTable: fe3d0f48  TableSize:  30.
    Image: regsvc.exe
```

Now use the **.context** command with this page directory base.

```
kd> .context 0011f000
```

This enables you to examine the address space in various ways. For example, here is the output of the **!peb** extension:

```
kd> !peb
PEB at 7FFDF000
    InheritedAddressSpace:    No
    ReadImageFileExecOptions: No
    BeingDebugged:            No
    ImageBaseAddress:         01000000
    Ldr.Initialized: Yes
    Ldr.InInitializationOrderModuleList: 71f40 . 77f68
    Ldr.InLoadOrderModuleList: 71ec0 . 77f58
    Ldr.InMemoryOrderModuleList: 71ec8 . 77f60
        01000000 C:\WINNT\system32\regsvc.exe
        77F80000 C:\WINNT\System32\ntdll.dll
        77DB0000 C:\WINNT\system32\ADVAPI32.dll
        77E80000 C:\WINNT\system32\KERNEL32.DLL
        77D40000 C:\WINNT\system32\RPCRT4.DLL
        77BE0000 C:\WINNT\system32\secur32.dll
    SubSystemData:     0
    ProcessHeap:       70000
    ProcessParameters: 20000
        WindowTitle:  'C:\WINNT\system32\regsvc.exe'
        ImageFile:    'C:\WINNT\system32\regsvc.exe'
        CommandLine:  'C:\WINNT\system32\regsvc.exe'
        DllPath:
'C:\WINNT\system32;.;C:\WINNT\System32;C:\WINNT\system;C:\WINNT;C:\WINNT\system32;C:\WINNT;C:\WINNT\System32\W
bem;C:\PROGRA~1\COMMON~1\AUTODE~1'
        Environment:  0x10000
```

.copysym (Copy Symbol Files)

12/5/2018 • 2 minutes to read • Edit Online

The **.copysym** command copies the current symbol files to the specified directory.

```
.copysym [/l] Path
```

Parameters

/l
Causes each symbol file to be loaded as it is copied.

Path
Specifies the directory to which the symbol files should be copied. Copies do not overwrite existing files.

Environment

Modes	user mode, kernel mode
Targets	live, crash dump
Platforms	all

Remarks

Many times, symbols are stored on a network. The symbol access can often be slow, or you may need to transport your debugging session to another computer where you no longer have network access. In such scenarios, the **.copysym** command can be used to copy the symbols you need for your session to a local directory.

.cordll (Control CLR Debugging)

12/5/2018 • 2 minutes to read • Edit Online

The **.cordll** command controls managed code debugging and the Microsoft .NET common language runtime (CLR).

```
.cordll [Options]
```

Parameters

Options
One or more of the following options:

-l (lower-case L)
Loads the CLR debugging modules.

-I **** *Module* (upper-case i)
Specifies the name or base address of the CLR module to be debugged. For more information, see Remarks.

-u
Unloads the CLR debugging modules.

-e
Enables CLR debugging.

-d
Disables CLR debugging.

-D
Disables CLR debugging and unloads the CLR debugging modules.

-N
Reloads the CLR debugging modules.

-lp **** *Path*
Specifies the directory path of the CLR debugging modules.

-se
Enables using the short name of the CLR debugging module, mscordacwks.dll.

-sd
Disables using the short name of the CLR debugging module, mscordacwks.dll. Instead, the debugger uses the long name of the CLR debugging module, mscordacwks_<spec>.dll. Turning off short name usage enables you to avoid having your local CLR used if you are concerned about mismatches.

-ve
Turns on verbose mode for CLR module loading.

-vd
Turns off verbose mode for CLR module loading.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

To debug a managed application, the debugger must load a data access component (DAC) that corresponds to the CLR that the application has loaded. However, in some cases, the application loads more than one CLR. In that case, you can use the **I** parameter to specify which DAC the debugger should load. Version 2 of the CLR is named Mscorwks.dll, and version 4 of the CLR is named Clr.dll. The following example shows how to specify that the debugger should load the DAC for version 2 (mscorwks).

```
.cordll -I mscorwks -lp c:\dacFolder
```

If you omit the **I** parameter, the debugger uses version 4 by default. For example, the following two commands are equivalent.

```
.cordll -lp c:\dacFolder
.cordll -I clr -lp c:\dacFolder
```

Sos.dll is a component that is used for debugging managed code. The current version of Debugging Tools for Windows does not include any version of sos.dll. For information about how to get sos.dll, see Getting the SOS Debugging Extension (sos.dll).

The **.cordll** command is supported in kernel-mode debugging. However, this command might not work unless the necessary memory is paged in.

See also

Debugging Managed Code Using the Windows Debugger

SOS Debugging Extension

.crash (Force System Crash)

12/5/2018 • 2 minutes to read • Edit Online

The **.crash** command causes the target computer to issue a bug check.

```
.crash
```

Environment

Modes	kernel mode only
Targets	live debugging only
Platforms	all

Additional Information

For an overview of related commands and a description of the options available after a system crash, see Crashing and Rebooting the Target Computer.

Remarks

This command will immediately cause the target computer to crash.

If you are already in a bug check handler, do not use **.crash**. Use **g (Go)** instead to continue execution of the handler, which will generate a crash dump.

A kernel-mode dump file will be written if crash dumps have been enabled. See Creating a Kernel-Mode Dump File for details.

.create (Create Process)

12/5/2018 • 2 minutes to read • Edit Online

The **.create** command creates a new target application.

```
.create [-premote RemoteOptions] [-f] CommandLine
```

Parameters

RemoteOptions
Specifies a process server to which to attach. The options are the same as those for the command line **-premote** option. See **Activating a Smart Client** for syntax details.

-f
Freezes all threads in all target applications, except in the new target being created. These threads will remain frozen until an exception occurs in the newly-created process. Note that an initial breakpoint qualifies as an exception. Individual threads can be unfrozen by using the **~u (Unfreeze Thread)** command.

CommandLine
Specifies the complete command line for the new process. *CommandLine* may contain spaces, and must not be surrounded with quotes. All text after the **.create** command is taken as part of *CommandLine*; this command cannot be followed with a semicolon and additional debugger commands.

Environment

Modes	user mode only
Targets	live debugging only
Platforms	all

Remarks

This command can be used when CDB is dormant, or if it is already debugging one or more processes. It cannot be used when WinDbg is dormant.

If this command is successful, the debugger will create the specified process the next time the debugger issues an execution command. If this command is used several times in a row, execution will have to be requested as many times as this command was used.

Multiple target processes will always be executed together, unless some of their threads are frozen or suspended.

If you wish to create a new process and freeze all your existing targets, use the -f option.

If the **-premote** option is used, the new process will be part of a new system. For details, see Debugging Multiple Targets.

.createdir (Set Created Process Directory)

12/5/2018 • 2 minutes to read • Edit Online

The **.createdir** command controls the starting directory and handle inheritance for any processes created by the debugger.

```
.createdir [-i | -I] [Path]
```

Parameters

-i
Causes processes created by the debugger to inherit handles from the debugger. This is the default.

-I
Prevents processes created by the debugger from inheriting handles from the debugger.

Path
Specifies the starting directory for all child processes created by any target process. If *Path* contains spaces, it must be enclosed in quotation marks.

Environment

Modes	user mode only
Targets	live debugging only
Platforms	all

Remarks

If **.createdir** is used with no parameters, the current starting directory and handle inheritance status are displayed.

If **.createdir** has never been used, any created process will use its usual default directory as its starting directory. If you have already set a path with **.createdir** and want to return to the default status, use **.createdir ""** with nothing inside the quotation marks.

The **.createdir** setting affects all processes created by **.create (Create Process)**. It also affects processes created by WinDbg's File | Open Executable menu command, unless the **Start directory** text box is used to override this setting.

.cxr (Display Context Record)

1/11/2019 • 2 minutes to read • Edit Online

The **.cxr** command displays the context record saved at the specified address. It also sets the register context.

```
.cxr [Options] [Address]
```

Parameters

Options
Can be any combination of the following options:

/f **** *Size*
Forces the context size to equal the value of *Size*, in bytes. This can be useful when the context does not match the actual target -- for example, when using an x86 context on a 64-bit target during *WOW64* debugging. If an invalid or inconsistent size is specified, the error "Unable to convert context to canonical form" will be displayed.

/w
Writes the current context to memory, and displays the address of the location where it was written.

Address
Address of the system context record.

Omitting the address does not display any context record information, but it does reset the register context.

Environment

Modes	user mode, kernel mode
Targets	live, crash dump
Platforms	all

Additional Information

For more information about the register context and other context settings, see Changing Contexts.

Remarks

The information from a context record can be used to assist in debugging a system halt where an unhandled exception has occurred and an exact stack trace is not available. The **.cxr** command displays the important registers for the specified context record.

This command also instructs the debugger to use the specified context record as the register context. After this command is executed, the debugger will have access to the most important registers and the stack trace for this thread. This register context persists until you allow the target to execute or use another register context command (**.thread**, **.ecxr**, **.trap** , or **.cxr** again). In user mode, it will also be reset if you change the current process or thread. See Register Context for details.

The **.cxr** command is often used to debug bug check 0x1E. For more information and an example, see **Bug Check 0x1E** (KMODE_EXCEPTION_NOT_HANDLED).

The **.cxr /w** command writes the context to memory and displays the address where it has been stored. This address can be passed to **.apply_dbp (Apply Data Breakpoint to Context)** if you need to apply data breakpoints to this context.

.dbgdbg (Debug Current Debugger)

12/5/2018 • 2 minutes to read • Edit Online

The **.dbgdbg** command launches a new instance of CDB; this new debugger takes the current debugger as its target.

```
.dbgdbg
```

Environment

Modes	user mode, kernel mode
Targets	live, crash dump
Platforms	all

Remarks

The **.dbgdbg** command is similar to the **CTRL+P (Debug Current Debugger)** control key. However, **.dbgdbg** is more versatile, because it can be used from WinDbg as well as KD and CDB, and it can be used to debug a debugging server on a remote computer.

.detach (Detach from Process)

1/3/2019 • 2 minutes to read • Edit Online

The **.detach** command ends the debugging session, but leaves any user-mode target application running.

```
.detach [ /h | /n ]
```

Parameters

/h
Any outstanding debug event will be continued and marked as handled. This is the default.

/n
Any outstanding debug event will be continued without being marked as handled.

Environment

For live user-mode debugging, this command is only supported in Windows XP and later versions of Windows.

Modes	user mode, kernel mode
Targets	live, crash dump
Platforms	all

Remarks

This command will end the debugging session in any of the following scenarios:

- When you are debugging a user-mode or kernel-mode dump file.
- When you are debugging a live user-mode target.
- When you are noninvasively debugging a user-mode target.

If you are only debugging a single target, the debugger will return to dormant mode after it detaches.

If you are debugging multiple targets, the debugging session will continue with the remaining targets.

.dml_flow (Unasemmble with Links)

12/5/2018 • 2 minutes to read • Edit Online

The **.dml_flow** command displays a disassembled code block and provides links that you can use to construct a code flow graph.

```
.dml_flow Start Target
```

Parameters

Start
The address of an instruction from which the target address can be reached.

Target
An address in the code block to be disassembled.

Remarks

Consider the call stack shown in the following example.

```
0: kd> kL
Child-SP          RetAddr           Call Site
fffff880`0335c688 fffff800`01b41f1c nt!IofCallDriver
fffff880`0335c690 fffff800`01b3b6b4 nt!IoSynchronousPageWrite+0x1cc
fffff880`0335c700 fffff800`01b4195e nt!MiFlushSectionInternal+0x9b8
...
```

Suppose you want to examine all code paths from the start of **nt!MiFlushSectionInternal** to the code block that contains the return adress, `fffff800`01b3b6b4`. The following command gets you started.

```
.browse .dml_flow nt!MiFlushSectionInternal fffff800`01b3b6b4
```

The output, in the Command Browser window, is shown in the following image.



The preceding image shows the code block that contains the target address, `fffff800`01b3b6b4`. There is only one

link (`fffff800`01b3b681`) at the top of the image. That indicates that there is only one code block from which the current code block can be reached. If you click the link, you will see that code block disassembled, and you will see links that enable you to further explore the code flow graph.

The two links at the bottom of the preceding image indicate that there are two code blocks that can be reached from the current code block.

See also

Debugger Markup Language Commands

uf (Unasemmble Function)

.dml_start (Display DML Starting Point)

12/5/2018 • 2 minutes to read • Edit Online

The **.dml_start** command displays output that serves as a starting point for exploration using commands that support Debugger Markup Language (DML).

```
.dml_start
.dml_start filename
```

Parameters

filename
The name of a DML file to be displayed as the starting output.

Using the Default Starting Output

If *filename* is omitted, the debugger displays a default DML starting output as illustrated in the following image.



Each line of output in the preceding example is a link that you can click to invoke other commands.

Providing a DML File

If you supply a path to a DML file, the file is used as the starting output. For example, suppose the file c:\MyFavoriteCommands.txt contains the following text and DML tags.

```
Display all device nodes.
   <link cmd="!devnode 0 1">!devnode 0 1</link>

Display all device nodes that are driven by a specified service.
Include child nodes in the display.
   <b>!devnode 0 1</b> <i>ServiceName</i>
   Example: <link cmd="!devnode 0 1 usbehci">!devnode 0 1 usbehci</link>

Explore device stacks, device objects, and driver objects.
   <b>!devstack</b>  List the device objects in a device stack.
   <b>!devobj</b>    Display information about a device object.
   <b>!drvobj</b>    Display information about a driver object.
```

The command **.dml_start c:\MyFavoriteCommands.txt** will display the file as shown in the following image.



Remarks

For information about DML tags that can be used in DML files, see dml.doc in the installation folder for Debugging Tools for Windows.

DML output often works well in the Command Browser window. To display a DML file in the Command Browser window, use **.browse .dml_start** *filename*.

See also

Debugger Markup Language Commands

.browse

.dump (Create Dump File)

12/5/2018 • 4 minutes to read • Edit Online

The **.dump** command creates a user-mode or kernel-mode crash dump file.

```
.dump Options FileName
.dump /?
```

Parameters

Options
Represents one or more of the following options

/o
Overwrites an existing dump file with the same name. If this option is not used and there is a file with the same file name, the dump file is not written.

/f[*FullOptions***]**
(Kernel mode:) Creates a complete memory dump.

(User mode:) Creates a *full user-mode dump*. For more information, see Varieties of User-Mode Dump Files. Despite their names, the largest minidump file actually contains more information than a full user-mode dump. For example, **.dump /mf** or **.dump /ma** creates a larger and more complete file than **.dump /f**. In user mode, **.dump /m[***MiniOptions***]** is always preferable to **.dump /f**.

You can add the following *FullOptions* to change the contents of the dump file; the option is case-sensitive.

FullOption	Effect
y	Adds AVX register information to the dump file.

/m[*MiniOptions***]**
Creates a *small memory dump* (in kernel mode) or a *minidump* (in user mode) For more information, see User-Mode Dump Files. If neither **/f** nor **/m** is specified, **/m** is the default.

In user mode, **/m** can be followed with additional *MiniOptions* specifying extra data that is to be included in the dump. If no *MiniOptions* are included, the dump will include module, thread, and stack information, but no additional data. You can add any of the following *MiniOptions* to change the contents of the dump file; they are case-sensitive.

MINIOPTION	EFFECT
a	Creates a minidump with all optional additions. The /ma option is equivalent to /mfFhut -- it adds full memory data, handle data, unloaded module information, basic memory information, and thread time information to the minidump. Any failure to read inaccessable memory results in termination of the minidump generation.

MINIOPTION	EFFECT
A	The /mA option is equivalent to /ma except that it ignores any failure to read inaccessable memory and continues generating the minidump.
f	Adds full memory data to the minidump. All accessible committed pages owned by the target application will be included.
F	Adds all basic memory information to the minidump. This adds a stream to the minidump that contains all basic memory information, not just information about valid memory. This allows the debugger to reconstruct the complete virtual memory layout of the process when the minidump is being debugged.
h	Adds data about the handles associated with the target application to the minidump.
u	Adds unloaded module information to the minidump. This is available only in Windows Server 2003 and later versions of Windows.
t	Adds additional thread information to the minidump. This includes thread times, which can be displayed by using the !runaway extension or the .ttime (Display Thread Times) command when debugging the minidump.
i	Adds secondary memory to the minidump. Secondary memory is any memory referenced by a pointer on the stack or backing store, plus a small region surrounding this address.
p	Adds process environment block (PEB) and thread environment block (TEB) data to the minidump. This can be useful if you need access to Windows system information regarding the application's processes and threads.
w	Adds all committed read-write private pages to the minidump.
d	Adds all read-write data segments within the executable image to the minidump.
c	Adds code sections within images.
r	Deletes from the minidump those portions of the stack and store memory that are not useful for recreating the stack trace. Local variables and other data type values are deleted as well. This option does not make the minidump smaller (because these memory sections are simply zeroed), but it is useful if you want to protect the privacy of other applications.
R	Deletes the full module paths from the minidump. Only the module names will be included. This is a useful option if you want to protect the privacy of the user's directory structure.

MINIOPTION	EFFECT
y	Adds AVX register information to the dump file.

Additional Information

For a description of kernel-mode dump files and an explanation of their use, see Kernel-Mode Dump Files. For a description of user-mode dump files and an explanation of their use, see User-Mode Dump Files.

Remarks

This command can be used in a variety of situations:

- During live user-mode debugging, this command directs the target application to generate a dump file, but the target application does not terminate.
- During live kernel-mode debugging, this command directs the target computer to generate a dump file, but the target computer does not crash.
- During crash dump debugging, this command creates a new crash dump file from the old one. This is useful if you have a large crash dump file and want to create a smaller one.

You can control what type of dump file will be produced:

- In kernel mode, to produce a complete memory dump, use the **/f** option. To produce a small memory dump, use the **/m** option (or no options). The .dump command cannot produce a kernel memory dump.
- In user mode, **.dump /m[***MiniOptions***]** is the best choice. Although "m" stands for "minidump", the dump files created by using this *MiniOption* can vary in size from very small to very large. By specifying the proper *MiniOptions* you can control exactly what information is included. For example, **.dump /ma** produces a dump with a great deal of information. The older command, **.dump /f**, produces a moderately large "standard dump" file and cannot be customized.

You cannot specify which process is dumped. All running processes will be dumped.

The **/xc**, **/xr**, **/xp**, and **/xt** options are used to store exception and context information in the dump file. This allows the **.ecxr (Display Exception Context Record)** command to be run on this dump file.

The following example will create a user-mode minidump, containing full memory and handle information:

```
0:000> .dump /mfh myfile.dmp
```

Handle information can be read by using the **!handle** extension command.

.dumpcab (Create Dump File CAB)

12/5/2018 • 2 minutes to read • Edit Online

The **.dumpcab** command creates a CAB file containing the current dump file.

```
.dumpcab [-a] CabName
```

Parameters

-a
Causes all currently loaded symbols to be included in the CAB file. For minidumps, all loaded images will be included as well. Use **lml** to determine which symbols and images are loaded.

CabName
The CAB file name, including extension. *CabName* can include an absolute or relative path; relative paths are relative to the directory in which the debugger was started. It is recommended that you choose the extension .cab.

Environment

Modes	user mode, kernel mode
Targets	crash dump
Platforms	all

Additional Information

For more details on crash dumps, see Crash Dump Files.

Remarks

This command can only be used if you are already debugging a dump file.

If you are debugging a live target and want to create a dump file and place it in a CAB, you should use the **.dump (Create Dump File)** command. Next, start a new debugging session with the dump file as its target, and use **.dumpcab**.

The **.dumpcab** command cannot be used to place multiple dump files into one CAB file.

.dvalloc (Allocate Memory)

12/5/2018 • 2 minutes to read • Edit Online

The **.dvalloc** command causes Windows to allocate additional memory to the target process.

```
.dvalloc [Options] Size
```

Parameters

Options
Can be any number of the following options:

/b **** *BaseAddress*
Specifies the virtual address of the beginning of the allocation.

/r
Reserves the memory in the virtual address space but does not actually allocate any physical memory. If this option is used, the debugger calls **VirtualAllocEx** with the *flAllocationType* parameter equal to MEM_RESERVE. If this option is not used, the value MEM_COMMIT | MEM_RESERVE is used. See the Microsoft Windows SDK for details.

Size
Specifies the amount of memory to be allocated, in bytes. The amount of memory available to the program will equal *Size*. The amount of memory actually used may be slightly larger, since it is always a whole number of pages.

Environment

Modes	user mode only
Targets	live debugging only
Platforms	all

Remarks

The **.dvalloc** command calls **VirtualAllocEx** to allocate new memory for the target process. The allocated memory permits reading, writing, and execution.

To free this memory, use **.dvfree (Free Memory)**.

.dvfree (Free Memory)

12/5/2018 • 2 minutes to read • Edit Online

The **.dvfree** command frees a memory allocation owned by the target process.

```
.dvfree [/d] BaseAddress Size
```

Parameters

/d
Decommits the allocation, but does not actually release the pages containing the allocation. If this option is used, the debugger calls **VirtualFreeEx** with the *dwFreeType* parameter equal to MEM_DECOMMIT. If this option is not used, the value MEM_RELEASE is used. See the Microsoft Windows SDK for details.

BaseAddress
Specifies the virtual address of the beginning of the allocation.

Size
Specifies the amount of memory to be freed, in bytes. The actual memory freed will always be a whole number of memory pages.

Environment

Modes	user mode only
Targets	live debugging only
Platforms	all

Remarks

The **.dvfree** command calls **VirtualFreeEx** to free an existing memory allocation. Unless the **/d** option is specified, the pages containing this memory are released.

This command can be used to free an allocation made by **.dvalloc (Allocate Memory)**. It can also be used to free any block of memory owned by the target process, but freeing memory that was not acquired through **.dvalloc** will naturally pose risks to the stability of the target process.

.echo (Echo Comment)

12/5/2018 • 2 minutes to read • Edit Online

The **.echo** command displays a comment string.

```
.echo String
.echo "String"
```

Parameters

String
Specifies the text to display. You can also enclose *String* in quotation marks ("). Regardless of whether you use quotation marks, *String* can contain any number of spaces, commas, and single quotation marks ('). If you enclose *String* in quotation marks, it can include semicolons, but not additional quotation marks. If you do not enclose *String* in quotation marks, it can include quotation marks in any location except the first character, but it cannot include semicolons.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

The **.echo** command causes the debugger to display *String* immediately after you enter the command.

An **.echo** command is ended if the debugger encounters a semicolon (unless the semicolon occurs within a quoted string). This restriction enables you to use **.echo** in complex constructions such as conditional breakpoints, as the following example shows.

```
0:000> bp `:143` "j (poi(MyVar)>5) '.echo MyVar Too Big'; '.echo MyVar Acceptable; gc' "
```

The **.echo** command also provides an easy way for users of debugging servers and debugging clients to communicate with one another. For more information about this situation, see Controlling a Remote Debugging Session.

The **.echo** command differs from the **$$ (Comment Specifier)** token and the *** (Comment Line Specifier)** token, because these tokens cause the debugger to ignore the input text without displaying it.

.echocpunum (Show CPU Number)

12/5/2018 • 2 minutes to read • Edit Online

The **.echocpunum** command turns on or turns off the display of the processor number when you are debugging a multiprocessor target computer.

```
.echocpunum 0
.echocpunum 1
.echocpunum
```

Parameters

0
Turns off the display of the processor number.

1
Turns on the display of the processor number.

Environment

Modes	Kernel mode only
Targets	Live debugging only
Platforms	All

Additional Information

For more information about how to debug multiprocessor computers, see Multiprocessor Syntax.

Remarks

If you use the **.echocpunum** command without any arguments, the display of processor numbers is turned on or off, and the new state is displayed.

By default, the display is turned off.

.echotime (Show Current Time)

12/5/2018 • 2 minutes to read • Edit Online

The **.echotime** command displays the current time.

```
.echotime
```

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

The **.echotime** command displays the time to the computer that the debugger is running on.

.echotimestamps (Show Time Stamps)

12/5/2018 • 2 minutes to read • Edit Online

The **.echotimestamps** command turns on or turns off the display of time stamp information.

```
.echotimestamps 0
.echotimestamps 1
.echotimestamps
```

Parameters

0
Turns off the display of time stamp information. This is the default behavior of the Debugger.

1
Turns on the display of time stamp information.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about **DbgPrint**, **KdPrint**, **DbgPrintEx**, and **KdPrintEx**, see The DbgPrint Buffer or see the Microsoft Windows Driver Kit (WDK) documentation.

Remarks

When you use the **.echotimestamps** command without parameters, the display of time stamps is turned on or off, and the new state is displayed.

If you turn on this display, the debugger shows time stamps for module loads, thread creations, exceptions, and other events.

The **DbgPrint**, **KdPrint**, **DbgPrintEx**, and **KdPrintEx** kernel-mode routines send a formatted string to a buffer on the host computer. The string is displayed in the Debugger Command window (unless you have disabled such printing). You can also display the formatted string by using the **!dbgprint** extension command.

When you use **.echotimestamps** to turn on the display of time stamps, the time and date of each comment in the **DbgPrint** buffer is displayed.

.ecxr (Display Exception Context Record)

12/5/2018 • 2 minutes to read • Edit Online

The **.ecxr** command displays the context record that is associated with the current exception.

```
.ecxr
```

Environment

Modes	User mode only
Targets	Crash dump only (minidumps only)
Platforms	All

Additional Information

For more information about the register context and other context settings, see Changing Contexts.

Remarks

The **.ecxr** command locates the current exception's context information and displays the important registers for the specified context record.

This command also instructs the debugger to use the context record that is associated with the current exception as the register context. After you run **.ecxr**, the debugger can access the most important registers and the stack trace for this thread. This register context persists until you enable the target to execute, change the current process or thread, or use another register context command (**.cxr** or **.ecxr**). For more information about register contexts, see Register Context.

The **.excr** command is a synonym command and has identical functionality.

See also

.excr

.effmach (Effective Machine)

3/5/2019 • 2 minutes to read • Edit Online

The **.effmach** command displays or changes the processor mode that the debugger uses.

```
.effmach [MachineType]
```

Parameters

MachineType
Specifies the processor type that the debugger uses for this session. If you omit this parameter, the debugger displays the current machine type.

You can enter one of the following machine types.

Machine type	Description
.	Use the processor mode of the target computer's native processor mode.
#	Use the processor mode of the code that is executing for the most recent event.
x86	Use an x86-based processor mode.
amd64	Use an x64-based processor mode.
ebc	Use an EFI byte code processor mode.
arm	Use an ARM64 processor mode.
chpe	Use a CHPE processor mode.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

The processor mode influences many debugger features:

- Which processor is used for stack tracing.
- Whether the process uses 32-bit or 64-bit pointers.
- Which processor's register set is active.

.enable_long_status (Enable Long Integer Display)

12/5/2018 • 2 minutes to read • Edit Online

The **.enable_long_status** command specifies whether the debugger displays long integers in decimal format or in the default radix.

```
.enable_long_status 0
.enable_long_status 1
```

Parameters

0
Displays all long integers in decimal format. This is the default behavior of the debugger.

1
Displays all long integers in the default radix.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

The **.enable_long_status** command affects the output of the **dt (Display Type)** command.

In WinDbg, **.enable_long_status** also affects the display in the Locals window and the

Watch window. These windows are automatically updated after you issue **.enable_long_status**.

This command affects only the display of long integers. To control whether standard integers are displayed in decimal format or the default radix, use the **.force_radix_output (Use Radix for Integers)** command.

Note The **dv (Display Local Variables)** command always displays long integers in the current number base.

To change the default radix, use the **n (Set Number Base)** command.

.enable_unicode (Enable Unicode Display)

12/5/2018 • 2 minutes to read • Edit Online

The **.enable_unicode** command specifies whether the debugger displays USHORT pointers and arrays as Unicode strings.

```
.enable_unicode 0
.enable_unicode 1
```

Parameters

0
Displays all 16-bit (USHORT) arrays and pointers as short integers. This is the default behavior of the debugger.

1
Displays all 16-bit (USHORT) arrays and pointers as Unicode strings.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

The **.enable_unicode** command affects the output of the **dt (Display Type)** command.

In WinDbg, the **.enable_unicode** command also affects the display in the Locals window and the Watch window. These windows are automatically updated after you issue **.enable_unicode**.

You can also select or clear **Display 16-bit values** as Unicode on the shortcut menu of the Locals or Watch window to specify the display for USHORT arrays and pointers.

See also

ds, dS (Display String)

.endpsrv (End Process Server)

12/5/2018 • 2 minutes to read • Edit Online

The **.endpsrv** command causes the current process server or KD connection server to close.

```
.endpsrv
```

Environment

You can use this command only when you are performing remote debugging through a process server or KD connection server.

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about these servers, see Process Servers (User Mode) or KD Connection Servers (Kernel Mode)

Remarks

The **.endpsrv** command terminates the process server or KD connection server currently connected to your smart client.

If you wish to terminate a process server or KD connection server from the computer on which it is running, use Task Manager to end the process (dbgsrv.exe or kdsrv.exe).

The **.endpsrv** command can terminate a process server or KD connection server, but it cannot terminate a debugging server. For information on how to do that, see Controlling a Remote Debugging Session.

.endsrv (End Debugging Server)

10/23/2019 • 2 minutes to read • Edit Online

The **.endsrv** command causes the debugger to cancel an active debugging server.

```
.endsrv ServerID
```

Parameters

ServerID
Specifies the ID of the debugging server.

Environment

You can use this command only when you are performing remote debugging through the debugger.

Modes	User mode only
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about remote debugging, see Remote Debugging Through the Debugger.

Remarks

You must issue the **.endsrv** command from the debugging server or from one of the debugging clients that are connected to the debugging server.

To determine the ID of a debugging server, use the **.servers (List Debugging Servers)** command.

The **.endsrv** command can terminate a debugging server, but it cannot terminate a process server or KD connection server. For information on how to end these servers, see Controlling a Process Server Session and Controlling a KD Connection Server Session. (There is, however, one exceptional case when **.endsrv** can end a process server that has been launched programmatically; for details, see **IDebugClient::StartProcessServer**.)

If you cancel a debugging server, you prevent any future debugging clients from attaching to the server. However, if you cancel a debugging server, you do not detach any clients that are currently attached through the server.

Consider the following situation. Suppose that you start some debugging servers, as the following example shows.

```
0:000> .server npipe:pipe=rabbit
Server started with 'npipe:pipe=rabbit'
0:000> .server tcp:port=7
Server started with 'tcp:port=7'
```

Then, you decide to use a password, as the following example shows.

```
0:000> .server npipe:pipe=tiger,password=hardtoguess
Server started with 'npipe:pipe=tiger,password=hardtoguess'
```

But the earlier servers are still running, so you should cancel them, as the following example shows.

```
0:000> .servers
0 - Debugger Server - npipe:Pipe=rabbit
1 - Debugger Server - tcp:Port=7
2 - Debugger Server - npipe:Pipe=tiger,Password=*
0:000> .endsrv 0
Server told to exit.  Actual exit may be delayed until
the next connection attempt.
0:000> .endsrv 1
Server told to exit.  Actual exit may be delayed until
the next connection attempt.
0:000> .servers
0 - <Disabled, exit pending>
1 - <Disabled, exit pending>
2 - Debugger Server - npipe:Pipe=tiger,Password=*
```

Finally, to make sure that nothing attached to your computer while the earlier servers were active, use the **.clients (List Debugging Clients)** command.

```
0:000> .clients
HotMachine\HostUser, last active Mon Mar 04 16:05:21 2002
```

Caution Using a password with TCP, NPIPE, or COM protocol offers only a small amount of protection, because the password is not encrypted. When you use a password together with a SSL or SPIPE protocol, the password is encrypted. If you want to establish a secure remote session, you must use the SSL or SPIPE protocol.

.enumtag (Enumerate Secondary Callback Data)

12/5/2018 • 2 minutes to read • Edit Online

The **.enumtag** command displays secondary bug check callback data and all data tags.

```
.enumtag
```

Environment

Modes	Kernel mode only
Targets	Live, crash dump
Platforms	All

Additional Information

For more information and for other ways of displaying this data, see Reading Bug Check Callback Data.

Remarks

You can use the **.enumtag** command only after a bug check has occurred or when you are debugging a kernel-mode crash dump file.

For each block of secondary bug check callback data, the **.enumtag** command displays the tag (in GUID format) and the data (as bytes and ASCII characters).

Consider the following example.

```
kd> .enumtag
{87654321-0000-0000-0000000000000000} - 0xf9c bytes
  4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00  MZ..............
  B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  ........@.......
  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
  ....
  00 00 00 00 00 00 00 00 00 00 00 00              ............

{12345678-0000-0000-0000000000000000} - 0x298 bytes
  F4 BF 7B 80 F4 BF 7B 80 00 00 00 00 00 00 00 00  ..{...{.........
 4B 44 42 47 98 02 00 00 00 20 4D 80 00 00 00 00  KDBG..... M.....
  54 A5 57 80 00 00 00 00 A0 50 5A 80 00 00 00 00  T.W......PZ.....
 40 01 08 00 18 00 00 00 BC 7D 50 80 00 00 00 00  @........}P.....
  ....
  00 00 00 00 00 00 00 00                          ........
```

In this example, the first batch of secondary data has a GUID){87654321-0000-0000-0000000000000000}) as its tag, and the second batch of data has a GUID ({12345678-0000-0000-0000000000000000}) as its tag. However, the data is not in a useful format.

.event_code (Display Event Code)

12/5/2018 • 2 minutes to read • Edit Online

The **.event_code** command displays the current event instructions.

```
.event_code
```

Environment

Modes	User mode, kernel mode
Targets	Live debugging only
Platforms	All

Remarks

The **.event_code** command displays the hexadecimal instructions at the current event's instruction pointer. The display includes up to 64 bytes of instructions if they are available.

.eventlog (Display Recent Events)

12/5/2018 • 2 minutes to read • Edit Online

The **.eventlog** command displays the recent Microsoft Win32 debug events, such as module loading, process creation and termination, and thread creation and termination.

```
.eventlog
```

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

The **.eventlog** command shows only 1024 characters.

The following example shows the **.eventlog** command.

```
0:000> .eventlog
0904.1014: Load module C:\Windows\system32\ADVAPI32.dll at 000007fe`fed80000
0904.1014: Load module C:\Windows\system32\RPCRT4.dll at 000007fe`fe8c0000
0904.1014: Load module C:\Windows\system32\GDI32.dll at 000007fe`fea00000
0904.1014: Load module C:\Windows\system32\USER32.dll at 00000000`76b10000
0904.1014: Load module C:\Windows\system32\msvcrt.dll at 000007fe`fe450000
0904.1014: Load module C:\Windows\system32\COMDLG32.dll at 000007fe`fecf0000
0904.1014: Load module C:\Windows\system32\SHLWAPI.dll at 000007fe`fe1f0000
0904.1014: Load module C:\Windows\WinSxS\amd64_microsoft.windows.common-controls_6595b6414
4ccf1df_6.0.6000.16386_none_1559f1c6f365a7fa\COMCTL32.dll at 000007fe`fbda0000
0904.1014: Load module C:\Windows\system32\SHELL32.dll at 000007fe`fd4a0000
0904.1014: Load module C:\Windows\system32\WINSPOOL.DRV at 000007fe`f77d0000
0904.1014: Load module C:\Windows\system32\ole32.dll at 000007fe`feb10000
0904.1014: Load module C:\Windows\system32\OLEAUT32.dll at 000007fe`feeb0000
Last event: Break instruction exception - code 80000003 (first chance)
```

.exepath (Set Executable Path)

12/5/2018 • 2 minutes to read • Edit Online

The **.exepath** command sets or displays the executable file search path.

```
.exepath[+] [Directory [; ...]]
```

Parameters

+
Specifies that the debugger should append the new directories to the previous executable file search path (instead of replacing the path).

Directory
Specifies one or more directories to put in the search path. If you do not specify *Directory*, the current path is displayed. You can separate multiple directories with semicolons.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

.expr (Choose Expression Evaluator)

12/5/2018 • 2 minutes to read • Edit Online

The **.expr** command specifies the default expression evaluator.

```
.expr /s masm
.expr /s c++
.expr /q
.expr
```

Parameters

/s masm
Changes the default expression type to Microsoft Assembler expression evaluator (MASM). This type is the default value when you start the debugger.

/s c++
Changes the default expression type to the C++ expression evaluator.

/q
Displays the list of possible expression types.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

When you use the **.expr** command without an argument, the debugger displays the current default expression type.

The **?? (Evaluate C++ Expression)** command, the Watch window, and the Locals window always use C++ expression syntax. All other commands and debugging information windows use the default expression evaluator.

For more information about how to control which syntax is used, see Evaluating Expressions. For more information about the syntax, see Numerical Expression Syntax.

.exptr (Display Exception Pointers)

12/5/2018 • 2 minutes to read • Edit Online

The **.exptr** command displays an EXCEPTION_POINTERS structure.

```
.exptr Address
```

Parameters

Address
Specifies the address of the EXCEPTION_POINTERS structure.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

.exr (Display Exception Record)

12/5/2018 • 2 minutes to read • Edit Online

The **.exr** command displays the contents of an exception record.

```
.exr Address
.exr -1
```

Parameters

Address
Specifies the address of the exception record. If you specify **-1** as the address, the debugger displays the most recent exception.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

The **.exr** command displays information that is related to an exception that the debugger encountered on the target computer. The information that is displayed includes the exception address, the exception code, the exception flags, and a variable list of parameters to the exception.

You can usually obtain the *Address* by using the **!pcr** extension.

The **.exr** command is often used to debug bug check 0x1E. For more information and an example, see **Bug Check 0x1E** (KMODE_EXCEPTION_NOT_HANDLED).

.excr (Display Exception Context Record)

12/5/2018 • 2 minutes to read • Edit Online

The **.excr** command displays the context record that is associated with the current exception.

```
.excr
```

Environment

Modes	User mode only
Targets	Crash dump only (minidumps only)
Platforms	All

Additional Information

For more information about the register context and other context settings, see Changing Contexts.

Remarks

The **.excr** command locates the current exception's context information and displays the important registers for the specified context record.

This command also instructs the debugger to use the context record that is associated with the current exception as the register context. After you run **.excr**, the debugger can access the most important registers and the stack trace for this thread. This register context persists until you enable the target to execute, change the current process or thread, or use another register context command (**.cxr** or **.excr**). For more information about register contexts, see Register Context.

The **.ecxr** command is a synonym command and has identical functionality.

See also

.ecxr

.extmatch (Display All Matching Extensions)

12/5/2018 • 2 minutes to read • Edit Online

The **.extmatch** command displays extension commands exported by the currently loaded extension DLLs that match the specified pattern.

```
.extmatch [Options] Pattern
```

Parameters

Options
Specifies the searching options. You can use one or more of the following options:

/e **** *ExtDLLPattern*
Limits the enumeration to extension commands exported by extension DLLs that match the specified pattern string. *ExtDLLPattern* is matched against the extension DLL filename.

/n
Excludes the extension name when the extensions are displayed. Thus, if this option is specified, then only the extension name itself will be displayed.

/D
Displays the output using Debugger Markup Language (DML). In the output, each listed extension is a link that you can click to get more information about the extension. Not all extension modules support DML.

Pattern
Specifies a pattern that the extension must contain. *Pattern*can contain a variety of wildcard characters and specifiers. For more information about the syntax, see String Wildcard Syntax.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

To display a list of loaded extension DLLs, use the **.chain** command.

Here is an example of this command, showing all the loaded extension DLLs that have an export named !help:

```
0:000> .extmatch help
!ext.help
!exts.help
!uext.help
!ntsdexts.help
```

The following example lists all extension commands beginning with the string "he" that are exported by extension DLLs whose names begin with the string "ex":

```
0:000> .extmatch /e ext* he*
!ext.heap
!ext.help
!exts.heap
!exts.help
```

The following example lists all extension commands, so we can see which ones support DML.



.extpath (Set Extension Path)

12/5/2018 • 2 minutes to read • Edit Online

The **.extpath** command sets or displays the extension DLL search path.

```
.extpath[+] [Directory[;...]]
```

Parameters

+
Signifies that the debugger should append new directories to the previous extension DLL search path (instead of replacing the path).

Directory
Specifies one or more directories to put in the search path. If you do not specify *Directory*, the current path is displayed. You can separate multiple directories with semicolons.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about the extension search path and loading extension DLLs, see Loading Debugger Extension DLLs.

Remarks

The extension DLL search path is reset to its default value at the start of each debugging session.

During live kernel-mode debugging, a reboot of the target computer results in the start of a new debugging session. So any changes that you make to the extension DLL search path during kernel-mode debugging will not persist across a reboot of the target computer.

The default value of the extension DLL search path contains all the extension paths known to the debugger and all the paths in the %PATH% environment variable. For example, suppose your %PATH% environment variable has a value of `C:\Windows\system32;C:\Windows`. Then the default value of the DLL extension search path might look like this.

```
0:000> .extpath
Extension search path is: C:\Program Files\Debugging Tools for Windows (x64)\WINXP;C:\Program Files\
Debugging Tools for Windows (x64)\winext;C:\Program Files\Debugging Tools for Windows (x64)\winext\
arcade;C:\Program Files\Debugging Tools for Windows (x64);C:\Program Files\Debugging Tools for
Windows (x64)\winext\arcade;C:\Windows\system32;C:\Windows
```

.f+, .f- (Shift Local Context)

12/5/2018 • 2 minutes to read • Edit Online

The **.f+** command shifts the frame index to the next frame in the current stack. The **.f-** command shifts the frame index to the previous frame in the current stack.

```
.f+
.f-
```

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about the local context and other context settings, see Changing Contexts. For more information about how to display local variables and other memory-related commands, see Reading and Writing Memory.

Remarks

The *frame* specifies the local context (scope) that the debugger uses to interpret local variables

The **.f+** and .f- commands are shortcuts for moving to the next and previous frames in the current stack. These commands are equivalent to the following **.frame** commands, but the **.f** commands are shorter for convenience:

- **.f+** is the same as **.frame @$frame + 1**.
- **.f-** is the same as **.frame @$frame - 1**.

The dollar sign ($) identifies the frame value as a pseudo-register. The at sign (@ causes the debugger to access the value more quickly, because it notifies the debugger that a string is a register or pseudo-register.

When an application is running, the meaning of local variables depends on the location of the program counter, because the scope of such variables extends only to the function that they are defined in. Unless you use an **.f+** or **.f-** command (or a **.frame** command), the debugger uses the scope of the current function (the current frame on the stack) as the local context.

The *frame number* is the position of the stack frame within the stack trace. You can view this stack trace by using the **k, kb, kc, kd, kp, kP, kv (Display Stack Backtrace)** command or the Calls window. The first line (the current frame) represents frame number 0. The subsequent lines represent frame numbers 1, 2, 3, and so on.

You can set the local context to a different stack frame to view new local variable information. However, the actual variables that are available depend on the code that is executed.

The debugger resets the local context to the scope of the program counter if any program execution occurs. The local context is reset to the top stack frame if the register context is changed.

.fiber (Set Fiber Context)

12/5/2018 • 2 minutes to read • Edit Online

The **.fiber** command specifies which fiber is used for the fiber context.

```
.fiber [Address]
```

Parameters

Address
Specifies the address of the fiber. If you omit this parameter or specify zero, the fiber context is reset to the current fiber.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about the process context, the register context, and the local context, see Changing Contexts.

.fiximports (Fix Target Module Imports)

12/5/2018 • 2 minutes to read • Edit Online

The **.fiximports** command validates and corrects all static import links for a target module.

```
.fiximports Module
```

Parameters

Module
Specifies the target module whose imports the debugger corrects. *Module* can contain a variety of wildcard characters and specifiers. For more information about the syntax, see String Wildcard Syntax. If you want to include spaces in *Module*, you must enclose the parameter in quotation marks.

Environment

Modes	User mode, kernel mode
Targets	Crash dump only (minidump only)
Platforms	All

Remarks

You can use the **.fiximports** command only when the target is a minidump that does not contain its own executable images.

When the debugger maps images for use as memory, the debugger does not automatically connect image imports to exporters. Therefore, instructions that refer to imports are disassembled in the same manner as in a live debugging session. You can use **.fiximports** to request that the debugger perform the appropriate import linking.

.flash_on_break (Flash on Break)

12/5/2018 • 2 minutes to read • Edit Online

The **.flash_on_break** command specifies whether the WinDbg taskbar entry flashes when WinDbg is minimized and the target breaks.

```
.flash_on_break on
.flash_on_break off
.flash_on_break
```

Parameters

on
Causes the WinDbg taskbar entry to flash if WinDbg is minimized and the target breaks into the debugger. This is the default behavior for WinDbg.

off
Prevents the WinDbg taskbar entry from flashing.

Environment

The **.flash_on_break** command is available only in WinDbg. You cannot use this command in script files.

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

If you use the **.flash_on_break** command without parameters, the debugger displays the current flash setting.

.fnent (Display Function Data)

12/5/2018 • 2 minutes to read • Edit Online

The **.fnent** command displays information about the function table entry for a specified function.

```
.fnent Address
```

Parameters

Address
Specifies the address of the function.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

The symbol search algorithm for the **.fnent** command is the same as that of the **ln (List Nearest Symbols)** command. The display first shows the nearest symbols. Then, the debugger displays the function entry for the first of these symbols.

If the nearest symbol is not in the function table, no information is displayed.

The following example shows a possible display.

```
0:001> .fnent 77f9f9e7
Debugger function entry 00b61f50 for:
(77f9f9e7)   ntdll!RtlpBreakWithStatusInstruction   |  (77f9fa98)   ntdll!DbgPrintReturnControlC

Params:    1
Locals:    0
Registers: 0

0:001> .fnent 77f9fa98
Debugger function entry 00b61f70 for:
(77f9fa98)   ntdll!DbgPrintReturnControlC   |  (77f9fb21)   ntdll!DbgPrompt

Non-FPO

0:001> .fnent 01005a60
No function entry for 01005a60
```

.fnret (Display Function Return Value)

12/5/2018 • 2 minutes to read • Edit Online

The **.fnret** command displays information about a function's return value.

```
.fnret [/s] Address [Value]
```

Parameters

/s
Sets the **$callret** pseudo-register equal to the return value that is being displayed, including type information.

Address
Specifies the address of the function.

Value
Specifies the return value to display. If you include *Value*, **.fnret** casts *Value* to the return type of the specified function and displays it in the format of the return type. If you omit *Value*, the debugger obtains the return value of the function from the return value registers.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

If you include the *Value* parameter, the **.fnret** command only casts this value to the proper type and displays the results.

If you omit *Value*, the debugger uses the return value registers to determine this value. If a function has returned more recently than the function that the *Address* parameter specifies, the value that is displayed will probably not be a value that this function returned.

.force_radix_output (Use Radix for Integers)

12/5/2018 • 2 minutes to read • Edit Online

The **.force_radix_output** command specifies whether integers are displayed in decimal format or in the default radix.

```
.force_radix_output 0
.force_radix_output 1
```

Parameters

0
Displays all integers (except for long integers) in decimal format. This is the default behavior for the Debugger.

1
Displays all integers (except for long integers) in the default radix.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

The **.force_radix_output** command affects the output of the **dt (Display Type)** command.

In WinDbg, **.force_radix_output** also affects the display in the Locals window and the Watch window. You can select or clear **Always display numbers in default radix** on the shortcut menu of the Locals or Watch window to have the same effect as **.force_radix_output**. These windows are automatically updated after you issue this command.

The **.force_radix_output** command affects only the display of standard integers. To specify whether long integers are displayed in decimal format or the default radix, use the **.enable_long_status (Enable Long Integer Display)** command.

To change the default radix, use the **n (Set Number Base)** command.

.force_tb (Forcibly Allow Branch Tracing)

12/5/2018 • 2 minutes to read • Edit Online

The **.force_tb** command forces the processor to trace branches early in the boot process.

```
.force_tb
```

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

Typically, branch tracing is enabled after the debugger initializes the processor control block (PRCB). This initialization occurs early in the boot process.

However, if you have to use the **tb (Trace to Next Branch)** command before this initialization, you can use the **.force_tb** command to enable branch tracing earlier. Use this command carefully because it can corrupt your processor state.

.formats (Show Number Formats)

12/5/2018 • 2 minutes to read • Edit Online

The **.formats** command evaluates an expression or symbol in the context of the current thread and process and displays it in multiple numeric formats.

```
.formats expression
```

Parameters

expression
Specifies the expression to evaluate. For more information about the syntax, see Numerical Expression Syntax.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

The evaluated expression is displayed in hexadecimal, decimal, octal, and binary formats, as well as single-precision and double-precision floating-point format. ASCII character formats are also displayed when the bytes correspond to standard ASCII characters. The expression is also interpreted as a time stamp if it is in the allowed range.

The following example shows a **.formats** command.

```
0:000> .formats 1c407e62
Evaluate expression:
  Hex:     1c407e62
  Decimal: 473988706
  Octal:   03420077142
  Binary:  00011100 01000000 01111110 01100010
  Chars:   .@~b
  Time:    Mon Jan 07 15:31:46 1985
  Float:   low 6.36908e-022 high 0
  Double:  2.34182e-315
```

The **Time** field displays a 32-bit value in CRT time stamp format and displays a 64-bit value in FILETIME format. You can distinguish these formats because the FILETIME format includes milliseconds and the CRT format does not.

See also

? (Evaluate Expression)

?? (Evaluate C++ Expression)

.fpo (Control FPO Overrides)

12/5/2018 • 2 minutes to read • Edit Online

The **.fpo** command controls the frame pointer omission (FPO) overrides.

```
.fpo -s [-fFlag] Address
.fpo -d Address
.fpo -x Address
.fpo -o Address
.fpo Address
```

Parameters

-s
Sets an FPO override at the specified address.

-f*Flag*
Specifies FPO flags for the override. You must not add a space between **-f** and *Flag*. If the flag takes an argument, you must add a space between the flag and that argument. If you want multiple flags, you must repeat the **-f** switch (for example, **-fb -fp 4 -fe**). You can use the **-f** switch only with **-s**. You can use one of the following values for *Flag*.

FLAG	EFFECT
b	Sets **fUseBP** equal to **TRUE**.
e	Sets **fUseSEH** equal to **TRUE**.
n	Sets **cbFrame** equal to FRAME_NONFPO. (By default, cbFrame is set to FRAME_FPO.)
l *Term*	Sets **cdwLocals** equal to *Term*. *Term* should specify the local DWORD count that you want.
p *Term*	Sets **cdwParams** equal to *Term*. *Term* should specify the parameter DWORD count that you want.
r *Term*	Sets **cbRegs** equal to *Term*. *Term* should specify the register count that you want.
s *Term*	Sets **cbProcSize** equal to *Term*. *Term* should specify the procedure size that you want.

FLAG	EFFECT
t *Term*	Sets **cbFrame** equal to *Term*. *Term* should specify one of the following frame types: • FRAME_FPO 0 • FRAME_TRAP 1 • FRAME_TSS 2 • FRAME_NONFPO 3

Address
Specifies the address where the debugger sets or removes the override or the address whose overrides the debugger should display. This address must be within a module in the current module list.

-d
Removes the FPO overrides at the specified address.

-x
Removes all FPO overrides within the module that contains the *Address* address.

-o
Displays all FPO overrides within the module that contains the *Address* address.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	all

Remarks

Without parameters, the **.fpo** command displays the FPO overrides for the specified address.

Some compilers (including Microsoft Visual Studio 6.0 and earlier versions) generate FPO information to indicate how the stack frame is set up. During stack walking, the debugger uses these FPO records to understand the stack. If the compiler generated incorrect FPO information, you can use the **.fpo** command to fix this problem.

.frame (Set Local Context)

12/5/2018 • 3 minutes to read • Edit Online

The **.frame** command specifies which local context (scope) is used to interpret local variables or displays the current local context.

```
.frame [/c] [/r] [FrameNumber]
.frame [/c] [/r] = BasePtr [FrameIncrement]
.frame [/c] [/r] = BasePtr StackPtr InstructionPtr
```

Parameters

/c
Sets the specified frame as the current local override context. This action allows a user to access the nonvolatile registers for any function in the call stack.

/r
Displays registers and other information about the specified local context.

FrameNumber
Specifies the number of the frame whose local context you want. If this parameter is zero, the command specifies the current frame. If you omit this parameter, this command displays the current local context.

BasePtr
Specifies the base pointer for the stack trace that is used to determine the frame, if you add an equal sign (=) after the command name (**.frame**). On an x86-based processor, you add another argument after *BasePtr* (which is interpreted as *FrameIncrement*) or two more arguments after *BasePtr* (which are interpreted as *InstructionPtr* and *StackPtr*).

FrameIncrement
(x86-based processor only)

Specifies an additional quantity of frames past the base pointer. For example, if the base pointer 0x0012FF00 is the address of frame 3, the command **.frame 12ff00** is equivalent to **.frame 3**, and **.frame 12ff00 2** is equivalent to **.frame 5**.

StackPtr
(x86-based processor only) Specifies the stack pointer for the stack trace that is used to determine the frame. If you omit *StackPtr* and *InstructionPtr*, the debugger uses the stack pointer that the **esp** register specifies and the instruction pointer that the **eip** register specifies.

InstructionPtr
(x86-based processor only) Specifies the instruction pointer for the stack trace that is used to determine the frame. If you omit *StackPtr* and *InstructionPtr*, the debugger uses the stack pointer that the **esp** register specifies and the instruction pointer that the **eip** register specifies.

Environment

Modes	User mode, kernel mode

Targets	Live, crash dump
Platforms	All

Additional Information

For more information about the local context and other context settings, see Changing Contexts. For more information about how to display local variables and other memory-related commands, see Reading and Writing Memory.

Remarks

When an application is running, the meaning of local variables depends on the location of the program counter, because the scope of such variables extends only to the function that they are defined in. If you do not use the **.frame** command, the debugger uses the scope of the current function (the current frame on the stack) as the local context.

To change the local context, use the **.frame** command and specify the frame number that you want.

The *frame number* is the position of the stack frame within the stack trace. You can view this stack trace with the **k (Display Stack Backtrace)** command or the Calls window. The first line (the current frame) is frame number 0. The subsequent lines represent frame numbers 1, 2, 3, and so on.

If you use the **n** parameter with the **k** command, the **k** command displays frame numbers together with the stack trace. These frame numbers are always displayed in hexadecimal form. On the other hand, the **.frame** command interprets its argument in the default radix, unless you override this setting with a prefix such as 0x. To change the default radix, use the **n (Set Number Base)** command.

You can set the local context to a different stack frame to enable you to view new local variable information. However, the actual variables that are available depend on the code that is being executed.

The local context is reset to the scope of the program counter if any application execution occurs. The local context is reset to the top stack frame if the register context is changed.

.help (Meta-Command Help)

12/5/2018 • 2 minutes to read • Edit Online

The **.help** command displays a list of all meta-commands.

```
.help
.help /D
```

Parameters

/D
Displays output using Debugger Markup Language. The output contains a list of links that you can click to see the meta-commands that beging with a particular letter.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

For more information about meta-commands, use the **.help** command. For more information about standard commands, use the **?** command. For more information about extension commands, use the **!help** extension.

.hh (Open HTML Help File)

12/5/2018 • 2 minutes to read • Edit Online

The **.hh** command opens the Debugging Tools for Windows documentation.

```
.hh [Text]
```

Parameters

Text
Specifies the text to find in the index of the Help documentation.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

You cannot use this command when you are performing remote debugging through Remote.exe.

Additional Information

For more information about the Help documentation, see Using the Help File.

Remarks

The **.hh** command opens the Debugging Tools for Windows documentation (Debugger.chm). If you specify *Text*, the debugger opens the **Index** pane in the documentation and searches for *Text* as a keyword in the index. If you do not specify *Text*, the debugger opens the **Contents** pane of the documentation.

.hideinjectedcode (Hide Injected Code)

12/5/2018 • 2 minutes to read • Edit Online

The **.hideinjectedcode** command turns on, or off, the hiding of injected code.

```
.hideinjectedcode { on | off }
.hideinjectedcode help
.hideinjectedcode
```

Parameters

on
Turns on the hiding of injected code.

off ****
Turns off the hiding of injected code

help
Displays help for this command.

Remarks

If you enter this command with no parameter, it displays help for the command.

.holdmem (Hold and Compare Memory)

12/5/2018 • 2 minutes to read • Edit Online

The **.holdmem** command saves memory ranges and compares them to other memory ranges.

```
.holdmem -a Range
.holdmem -d { Range | Address }
.holdmem -D
.holdmem -o
.holdmem -c Range
```

Parameters

-a **** *Range*
Specifies the memory range to save. For more information about the syntax, see Address and Address Range Syntax.

-d **** { *Range* | *Address* }
Specifies memory ranges to delete. If you specify *Address*, the debugger deletes any saved range that contains that address. If you specify *Range*, the debugger deletes any saved range that overlaps with *Range*. For more information about the syntax, see Address and Address Range Syntax.

-D
Deletes all saved memory ranges.

-o
Displays all saved memory ranges.

-c *Range*
Compares the specified range to all saved memory ranges. For more information about the syntax, see Address and Address Range Syntax.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about how to manipulate memory and a description of other memory-related commands, see Reading and Writing Memory.

Remarks

The **.holdmem** command compares memory ranges byte-for-byte.

If any of the specified memory locations do not exist in the virtual address space, the command returns an error.

.idle_cmd (Set Idle Command)

12/5/2018 • 2 minutes to read • Edit Online

The **.idle_cmd** command sets the *idle command*. This is a command that is executed whenever control returns from the target to the debugger. For example, when the target reaches a breakpoint, this command executes.

```
.idle_cmd
.idle_cmd String
.idle_cmd /d
```

Parameters

String
Specifies the string to which the idle command should be set.

/d
Clears the idle command.

Environment

This command cannot be used in script files.

Modes	user mode, kernel mode
Targets	live, crash dump
Platforms	all

Remarks

When **.idle_cmd** is used with no parameters it displays the current idle command.

In WinDbg, idle commands are stored in workspaces.

Here is an example. The idle command is set to **r eax**. Then, because the debugger is already idle, this command immediately executes, displaying the **eax** register:

```
windbg> .idle_cmd r eax
Execute when idle: r eax
eax=003b0de8
```

.ignore_missing_pages (Suppress Missing Page Errors)

12/5/2018 • 2 minutes to read • Edit Online

The **.ignore_missing_pages** command suppresses the error messages when a Kernel Memory Dump has missing pages.

```
.ignore_missing_pages 0
.ignore_missing_pages 1
.ignore_missing_pages
```

Parameters

0

Displays all missing page errors while debugging a Kernel Memory Dump. This is the default behavoir of the debugger.

1

Suppresses the display of missing page errors while debugging a Kernel Memory Dump.

Environment

Modes	Kernel mode only
Targets	Dump file debugging only
Platforms	All

Additional Information

For more information about how to debug these dump files, see Kernel Memory Dump.

Remarks

Without parameters, **.ignore_missing_pages** displays the current status of this setting.

.inline (Toggle Inline Function Debugging)

12/5/2018 • 2 minutes to read • Edit Online

The **.inline** command enables or disables inline function debugging.

```
.inline 0
.inline 1
```

Parameters

0

Disables inline function debugging.

1

Enables inline function debugging.

See also

Debugging Optimized Code and Inline Functions

.imgscan (Find Image Headers)

12/5/2018 • 2 minutes to read • Edit Online

The **.imgscan** command scans virtual memory for image headers.

```
.imgscan [Options]
```

Parameters

Options
Any of the following options:

/r **** *Range*
Specifies the range to search. For more information about the syntax, see Address and Address Range Syntax. If you specify only one address, the debugger searches a range that begins at that address and extends 0x10000 bytes.

/l
Loads module information for any image header that is found.

/v
Displays verbose information.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

If you do not use the **/r** parameter, the debugger searches all virtual memory regions.

The **.imgscan** command displays any image headers that it finds and the header type. Header types include Portable Executable (PE) headers and Microsoft MS-DOS MZ headers.

The following example shows the **.imgscan** command.

```
0:000> .imgscan
MZ at 00400000, prot 00000002, type 01000000 - size 2d000
MZ at 77f80000, prot 00000002, type 01000000 - size 7d000
  Name: ntdll.dll
MZ at 7c570000, prot 00000002, type 01000000 - size b8000
  Name: KERNEL32.dll
```

.jdinfo (Use JIT_DEBUG_INFO)

12/5/2018 • 4 minutes to read • Edit Online

The **.jdinfo** command uses a JIT_DEBUG_INFO structure as the source of the exception and context for just in time (JIT) debugging. The address to the structure is passed to the **.jdinfo** command using the %p parameter that is specified in the AeDebug registry entry.

For more information about the registry keys used, see Enabling Postmortem Debugging. For more information about register contexts, see Changing Contexts.

```
.jdinfo Address
```

Parameters

Address
Specifies the address of the JIT_DEBUG_INFO structure. The address to the structure is passed to the **.jdinfo** command using the %p parameter that is specified in the AeDebug registry entry.

Environment

Modes	User mode
Targets	Live, crash dump
Platforms	All

Example

This example show how the AeDebug registry entry can be configured to use the WinDbg can be used as the JIT debugger.

```
Debugger = "Path\WinDbg.EXE -p %ld -e %ld -c ".jdinfo 0x%p"
```

Then, when a crash occurs, the configured JIT debugger is invoked and the %p parameter is used to pass the address of the JIT_DEBUG_INFO structure to the **.jdinfo** command that is executed after the debugger is started.

```
nMicrosoft (R) Windows Debugger Version 10.0.10240.9 AMD64
Copyright (c) Microsoft Corporation. All rights reserved.

*** wait with pending attach
Executable search path is:
...
ModLoad: 00000000`68a20000 00000000`68ac3000
C:\WINDOWS\WinSxS\amd64_microsoft.vc90.crt_1fc8b3b9a1e18e3b_9.0.30729.9247_none_08e394a1a83e212f\MSVCR90.dll
(153c.5d0): Break instruction exception - code 80000003 (first chance)
Processing initial command '.jdinfo 0x00000000003E0000'
ntdll!DbgBreakPoint:
00007ffc`81a986a0 cc              int     3
0:003> .jdinfo 0x00000000003E0000
----- Exception occurred on thread 0:15c8
ntdll!ZwWaitForMultipleObjects+0x14:
00007ffc`81a959a4 c3              ret

----- Exception record at 00000000`003e0028:
ExceptionAddress: 00007ff791d81014 (CrashAV_x64!wmain+0x0000000000000014)
   ExceptionCode: c0000005 (Access violation)
  ExceptionFlags: 00000000
NumberParameters: 2
   Parameter[0]: 0000000000000001
   Parameter[1]: 0000000000000000
Attempt to write to address 0000000000000000

----- Context record at 00000000`003e00c0:
rax=0000000000000000 rbx=0000000000000000 rcx=00007ffc81a954d4
rdx=0000000000000000 rsi=0000000000000000 rdi=0000000000000001
rip=00007ff791d81014 rsp=00000000006ff8b0 rbp=0000000000000000
 r8=00000000006ff808  r9=0000000000000000 r10=0000000000000000
r11=0000000000000000 r12=0000000000000000 r13=0000000000000000
r14=0000000000000000 r15=0000000000000000
iopl=0         nv up ei pl zr na po nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010246
CrashAV_x64!wmain+0x14:
00007ff7`91d81014 45891b          mov     dword ptr [r11],r11d ds:00000000`00000000=????????
```

Remarks

The **.jdinfo** command uses the **AeDebug** registry information introduced in Windows Vista. For more information about the registry keys used, see Enabling Postmortem Debugging. The **.jdinfo** command takes the address of a JIT_DEBUG_INFO that the system set up for **AeDebug** and sets the context to the exception that caused the crash.

You can use the **.jdinfo** command instead of **-g** in **AeDebug** to have your debugger set to the **AeDebug** state without requiring execution.

This state can be advantageous, because under usual conditions, when a user-mode exception occurs, the following sequence occurs:

1. The Microsoft Windows operating system halts execution of the application.
2. The postmortem debugger is started.
3. The debugger attaches to the application.
4. The debugger issues a "Go" command. (This command is caused by the **-g** in the **AeDebug** key.)
5. The target attempts to execute and may or may not encounter the same exception.
6. This exception breaks into the debugger.

There are several problems that can occur because of these events:

- Exceptions do not always repeat, possibly because of a transient condition that no longer exists when the exception is restarted.
- Another event, such as a different exception, might occur. There is no way of knowing whether it is identical to the original event.
- Attaching a debugger involves injecting a new thread, which can be blocked if a thread is holding the loader lock. Injecting a new thread can be a significant disturbance of the process.

If you use **-c .jdinfo** instead of **-g** in your **AeDebug** key, no execution occurs. Instead, the exception information is retrieved from the JIT_DEBUG_INFO structure using the %p variable.

For example, consider the following **AeDebug** key.

```
ntsd -p %ld -e %ld -c ".jdinfo 0x%p"
```

The following example is even less invasive. The **-pv** switch causes the debugger to attach noninvasively, which does not inject any new threads into the target.

```
ntsd -pv -p %ld -e %ld -c ".jdinfo 0x%p"
```

If you use this noninvasive option, exiting the debugger does not end the process. You can use the **.kill (Kill Process)** command to end the process.

If you want to use this for dump file debugging, you should use **.dump /j** to add the JIT_DEBUG_INFO structure to your dump file, when the dump file is created.

The JIT_DEBUG_INFO structure is defined as follows.

```
typedef struct _JIT_DEBUG_INFO {
    DWORD dwSize;
    DWORD dwProcessorArchitecture;
    DWORD dwThreadID;
    DWORD dwReserved0;
    ULONG64 lpExceptionAddress;
    ULONG64 lpExceptionRecord;
    ULONG64 lpContextRecord;
} JIT_DEBUG_INFO, *LPJIT_DEBUG_INFO;
```

You can use the dt command to display the JIT_DEBUG_INFO structure.

```
0: kd> dt JIT_DEBUG_INFO
nt!JIT_DEBUG_INFO
   +0x000 dwSize           : Uint4B
   +0x004 dwProcessorArchitecture : Uint4B
   +0x008 dwThreadID       : Uint4B
  +0x00c dwReserved0      : Uint4B
   +0x010 lpExceptionAddress : Uint8B
   +0x018 lpExceptionRecord : Uint8B
   +0x020 lpContextRecord  : Uint8B
```

Viewing the Exception Record, Call Stack and LastEvent Using WinDbg

After the .jdinfo command has been used to set the context to the moment of failure, you can view the exception record returned by .jdinfo, the call stack and the lastevent, as shown below, to investigate cause.

```
0:000> .jdinfo  0x00000000003E0000
----- Exception occurred on thread 0:15c8
ntdll!NtWaitForMultipleObjects+0x14:
00007ffc`81a959a4 c3              ret

----- Exception record at 00000000`003e0028:
ExceptionAddress: 00007ff791d81014 (CrashAV_x64!wmain+0x0000000000000014)
   ExceptionCode: c0000005 (Access violation)
  ExceptionFlags: 00000000
NumberParameters: 2
   Parameter[0]: 0000000000000001
   Parameter[1]: 0000000000000000
Attempt to write to address 0000000000000000
...

0:000> k
  *** Stack trace for last set context - .thread/.cxr resets it
# Child-SP          RetAddr           Call Site
00 00000000`006ff8b0 00007ff7`91d811d2 CrashAV_x64!wmain+0x14 [c:\my\my_projects\crash\crashav\crashav.cpp @
14]
01 00000000`006ff8e0 00007ffc`7fa38364 CrashAV_x64!__tmainCRTStartup+0x11a
[f:\dd\vctools\crt_bld\self_64_amd64\crt\src\crtexe.c @ 579]
02 00000000`006ff910 00007ffc`81a55e91 KERNEL32!BaseThreadInitThunk+0x14
03 00000000`006ff940 00000000`00000000 ntdll!RtlUserThreadStart+0x21

0:000> .lastevent
Last event: 153c.5d0: Break instruction exception - code 80000003 (first chance)
  debugger time: Thu Sep  8 12:55:08.968 2016 (UTC - 7:00)
```

.kdfiles (Set Driver Replacement Map)

12/5/2018 • 2 minutes to read • Edit Online

The **.kdfiles** command reads a file and uses its contents as the driver replacement map.

```
.kdfiles MapFile
.kdfiles -m OldDriver NewDriver
.kdfiles -s SaveFile
.kdfiles -c
.kdfiles
```

Parameters

MapFile
Specifies the driver replacement map file to read.

-m
Adds a driver replacement association to the current association list.

OldDriver
Specifies the path and file name of the previous driver on the target computer. The syntax for *OldDriver* is the same as that of the first line after **map** in a driver replacement file. For more information about this syntax, see Mapping Driver Files.

NewDriver
Specifies the path and file name of the new driver. This driver can be on the host computer or at some other network location. The syntax for *NewDriver* is the same as that of the second line after **map** in a driver replacement file. For more information about this syntax, see Mapping Driver Files.

-s
Creates a file and writes the current driver replacement associations to that file.

SaveFile
Specifies the name of the file to create.

-c
Deletes the existing driver replacement map. (This option does not alter the map file itself. Instead, this option clears the debugger's current map settings.)

Environment

You can use the **.kdfiles** command in Microsoft Windows XP and later versions of Windows. If you use this command in earlier versions of Windows, the command has no effect and does not generate an error.

Modes	Kernel mode only
Targets	Live debugging only
Platforms	x86-based and Itanium-based processors only

Additional Information

For more information about and examples of driver replacement and the replacement of other kernel-mode modules, a description of the format for driver replacement map files, and restrictions for using this feature, see Mapping Driver Files.

Remarks

If you use the **.kdfiles** command without parameters, the debugger displays the path and name of the current driver replacement map file and the current set of replacement associations.

When you run this command, the specified *MapFile*file is read. If the file is not found or if it does not contain text in the proper format, the debugger displays a message that states, "Unable to load file associations".

If the specified file is in the correct driver replacement map file format, the debugger loads the file's contents and uses them as the driver replacement map. This map remains until you exit the debugger, or until you issue another **.kdfiles** command.

After the file has been read, the driver replacement map is not affected by subsequent changes to the file (unless these changes are followed by another **.kdfiles** command).

Requirements

Version	Supported in Windows XP and later versions of the Windows operating system.

.kdtargetmac (Display Target MAC Address)

12/5/2018 • 2 minutes to read • Edit Online

Display Target MAC Address

```
.kdtargetmac
```

Parameters

Environment

Modes	kernel mode only
Targets	live debugging only
Platforms	all

Additional Information

Use the .kdtargetmac command to display the MAC (media access control) address of the target system.

```
0: kd> .kdtargetmac

The target machine MAC address in open-device format is: XXXXXXXXXXXX
```

The .kdtargetmac is command is available if KDNET is enabled on the target system. Use the BCDEdit command with the /dbgsettings option to display the configuration on the target system. A debugtype of *NET* indicates that KDNET is configured.

```
C:\WINDOWS\system32>bcdedit /dbgsettings
key                     1.2.3.4
debugtype               NET
hostip                  192.168.1.10
port                    50000
dhcp                    Yes
The operation completed successfully.
```

For more information, see Setting Up KDNET Network Kernel Debugging Manually.

Remarks

Knowing the MAC address of the target system can be useful for network tracing and other utilities.

.kframes (Set Stack Length)

12/5/2018 • 2 minutes to read • Edit Online

The **.kframes** command sets the default length of a stack trace display.

```
.kframes FrameCountDefault
```

Parameters

FrameCountDefault
Specifies the number of stack frames to display when a stack trace command is used.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

You can use the **.kframes** command to set the default length of a stack trace display. This length controls the number of frames that the **k, kb, kp, kP, and kv** commands display and the number of DWORD_PTRs that the **kd** command displays.

You can override this default length by using the *FrameCount* or *WordCount* parameters for these commands.

If you never issue the **.kframes** command, the default count is 20 (0x14).

See also

k, kb, kc, kd, kp, kP, kv (Display Stack Backtrace)

.kill (Kill Process)

12/5/2018 • 2 minutes to read • Edit Online

In user mode, the **.kill** command ends a process that is being debugged.

In kernel mode, the **.kill** command ends a process on the target computer.

User-Mode Syntax

```
.kill [ /h | /n ]
```

Kernel-Mode Syntax

```
.kill Process
```

Parameters

/h
(User mode only) Any outstanding debug event will be continued and marked as handled. This is the default.

/n
(User mode only) Any outstanding debug event will be continued without being marked as handled.

Process
Specifies the address of the process to be terminated. If *Process* is omitted or zero, the default process for the current system state will be terminated.

Environment

In kernel mode, this command is supported on Microsoft Windows Server 2003 and later versions of Windows.

Modes	user mode, kernel mode
Targets	live debugging only
Platforms	all

Remarks

In user mode, this command ends a process that is being debugged. If the debugger is attached to a child process, you can use **.kill** to end the child process without ending the parent process. For more information, see Examples.

In kernel mode, this command schedules the selected process on the target computer for termination. The next time that the target can run (for example, by using a **g (Go)** command), the specified process is ended.

You cannot use this command during local kernel debugging.

Examples

Using .childdbg

Suppose you attach a debugger to parent process (Parent.exe) before it creates a child process. You can enter the command **.childdbg 1** to tell the debugger to attach to any child process that the parent creates.

```
1:001> .childdbg 1
Processes created by the current process will be debugged
```

Now let the parent process run, and break in after it has created the child process. Use the **| (Process Status)** command to see the process numbers for the parent and child processes.

```
0:002> |*
.  0    id: 7f8 attach  name: C:\Parent\x64\Debug\Parent.exe
   1    id: 2d4 child   name: notepad.exe
```

In the preceding output, the number of the child process (notepad.exe) is 1. The dot (.) at the beginning of the first line tells us that the parent process is the current process. To make the child process the current process, enter **|1s**.

```
0:002> |1s
...
1:001> |*
#  0    id: 7f8 attach  name: C:\Parent\x64\Debug\Parent.exe
.  1    id: 2d4 child   name: notepad.exe
```

To kill the child process, enter the command **.kill**. The parent process continues to run.

```
1:001> .kill
Terminated.  Exit thread and process events will occur.
1:001> g
```

Using the -o parameter

When you start WinDbg or CDB, you can use the **-o** parameter to tell the debugger that it should attach to child processes. For example, the following command starts WinDbg, which starts and attaches to Parent.exe. When Parent.exe creates a child process, WinDbg attaches to the child process.

windbg -g -G -o Parent.exe

For more information, see **WinDbg Command-Line Options** and **CDB Command-Line Options**.

Requirements

Version	Versions:(Kernel mode) Supported in Windows Server 2003 and later.

.lastevent (Display Last Event)

12/5/2018 • 2 minutes to read • Edit Online

The **.lastevent** command displays the most recent exception or event that occurred.

```
.lastevent
```

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about exceptions and events, see Controlling Exceptions and Events.

Remarks

Breaking into the debugger always creates an exception. There is always a *last event* when the debugger accepted command input. If you break into the debugger by using **CTRL+C**, CTRL+BREAK, or Debug | Break, an exception code of 0x80000003 is created.

.lines (Toggle Source Line Support)

12/5/2018 • 2 minutes to read • Edit Online

The **.lines** command enables or disables support for source-line information.

```
.lines [-e|-d|-t]
```

Parameters

-e
Enables source line support.

-d
Disables source line support.

-t
Turns source line support on or off. If you do not specify parameters for **.lines**, the default behavior of the **.lines** command is this switching of source line support.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about source debugging and related commands, see Debugging in Source Mode.

Remarks

You must enable source line support before you can perform source-level debugging. This support enables the debugger to load source line symbols.

You can enable source line support by using the **.lines** command or the -lines command-line option. If source line support is already enabled, using the **.lines** command disables this support.

By default, if you do not use the **.lines** command, WinDbg turns on source line support, and console debuggers (KD, CDB, NTSD) turn off the support. For more information about how to change this setting, see Setting Symbol Options.

.load, .loadby (Load Extension DLL)

12/5/2018 • 2 minutes to read • Edit Online

The **.load** and **.loadby** commands load a new extension DLL into the debugger.

```
.load DLLName
!DLLName.load
.loadby DLLName ModuleName
```

Parameters

DLLName
Specifies the debugger extension DLL to load. If you use the **.load** command, *DLLName* should include the full path. If you use the **.loadby** command, *DLLName* should include only the file name.

ModuleName
Specifies the module name of a module that is located in the same directory as the extension DLL that *DLLName* specifies.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about how to load, unload, and control extensions, see Loading Debugger Extension DLLs.

Remarks

When you use the **.load** command, you must specify the full path.

When you use the **.loadby** command, you do not specify the path. Instead, the debugger finds the module that the *ModuleName* parameter specifies, determines the path of that module, and then uses that path when the debugger loads the extension DLL. If the debugger cannot find the module or if it cannot find the extension DLL, you receive an error message that specifies the problem. There does not have to be any relationship between the specified module and the extension DLL. Using the **.loadby** command is therefore simply a way to avoid typing a long path.

After the **.load** or **.loadby** command has been completed, you can access the commands that are stored in the loaded extension.

To load an extension DLL, you can do one of the following:

- Use the **.load** or **.loadby** command.
- Execute an extension by issuing the full **!***DLLName*.*ExtensionCommand* syntax. If the debugger has not

yet loaded *DLLName*.dll, it loads the DLL at this point.

.locale (Set Locale)

12/5/2018 • 2 minutes to read • Edit Online

The **.locale** command sets the locale or displays the current locale.

```
.locale [Locale]
```

Parameters

Locale
Specifies the locale that you want. If you omit this parameter, the debugger displays the current locale.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about locale, see the **setlocale** routine reference page.

Remarks

The locale controls how Unicode strings are displayed.

The following examples show the **.locale** command.

```
kd> .locale
Locale: C

kd> .locale E
Locale: English_United States.1252

kd> .locale c
Locale: Catalan_Spain.1252

kd> .locale C
Locale: C
```

.logappend (Append Log File)

12/5/2018 • 2 minutes to read • Edit Online

The **.logappend** command appends a copy of the events and commands from the Debugger Command window to the specified log file.

```
.logappend [/u] [FileName]
```

Parameters

/u
Writes the log file in Unicode format. If you omit this parameter, the debugger writes the log file in ASCII (ANSI) format.

Note When you are appending to an existing log file, you should use the **/u** parameter only if you created the log file by using the **/u** option. Otherwise, your log file will contain ASCII and Unicode characters, which might make it more difficult to read.

FileName
Specifies the name of the log file. You can specify a full path or only the file name. If the file name contains spaces, enclose *FileName* in quotation marks. If you do not specify the path, the debugger uses the current directory. If you omit *FileName*, the debugger names the file Dbgeng.log.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

If you already have a log file open when you run the **.logappend** command, the debugger closes the log file. If you specify the name of a file that already exists, the debugger appends new information to the file. If the file does not exist, the debugger creates it.

.logclose (Close Log File)

12/5/2018 • 2 minutes to read • Edit Online

The **.logclose** command closes any open log file.

```
.logclose
```

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

.logfile (Display Log File Status)

12/5/2018 • 2 minutes to read • Edit Online

The **.logfile** command determines whether a log file exists and displays the file's status.

```
.logfile
```

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

.logopen (Open Log File)

12/5/2018 • 2 minutes to read • Edit Online

The **.logopen** command sends a copy of the events and commands from the Debugger Command window to a new log file.

```
.logopen [Options] [FileName]
.logopen /d
```

Parameters

Options
Any of the following options:

/t
Appends the process ID with the current date and time to the log file name. This data is inserted after the file name and before the file name extension.

/u
Writes the log file in Unicode format. If you omit this option, the debugger writes the log file in ASCII (ANSI) format.

FileName
Specifies the name of the log file. You can specify a full path or only the file name. If the file name contains spaces, enclose *FileName* in quotation marks. If you do not specify a path, the debugger uses the current directory. If you omit *FileName*, the debugger names the file Dbgeng.log.

/d
Automatically chooses a file name based on the name of the target process or target computer and the state of the target. The file always has the .log file name extension.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

If you already have a log file open when you run the **.logopen** command, the debugger closes it. If you specify a file name that already exists, the file's contents are overwritten.

The **.logopen /t** command appends the process ID, date, and time to the log file name. In the following example, the process ID in hexadecimal is 0x02BC, the date is February 28, 2005, and the time is 9:05:50.935.

```
0:000> .logopen /t c:\logs\mylogfile.txt
Opened log file 'c:\logs\mylogfile_02BC_2005-02-28_09-05-50-935.txt'
```

.netsyms (Disable Network Symbol Loading)

12/5/2018 • 2 minutes to read • Edit Online

Use the .netsyms command to allow or disallow loading symbols from a network path.

Syntax

.netsyms *{yes|no}*

Parameters

yes
Enables network symbol loading. This is the default.

no
Disables network symbol loading.

Environment

Modes	user mode, kernel mode
Targets	live, crash dump
Platforms	all

Remarks

Use .netsyms with no argument to display the current state of this setting.

Use **!sym noisy** or the *-n* **WinDbg Command-Line Option** to display additional detail as symbols are loaded.

.netuse (Control Network Connections)

12/5/2018 • 2 minutes to read • Edit Online

The **.netuse** command adds a connection to a network share.

```
.netuse /a "[Local]" "Remote" "[User]" "[Password]"
```

Parameters

/a
Adds a new connection. You must always use the **/a** switch.

Local
Specifies the drive letter to use for the connection. You must enclose *Local* in quotation marks. If you omit this parameter, you must include an empty pair of quotation marks as the parameter.

Remote
Specifies the UNC path of the share that is being connected. You must enclose *Remote* in quotation marks.

User
Specifies the user name of an account that is authorized to establish the connection. You must enclose *User* in quotation marks. If you omit this parameter, you must include an empty pair of quotation marks as the parameter.

Password
Specifies the password that is associated with the *User* account. You must enclose *Password* in quotation marks. If you omit this parameter, you must include an empty pair of quotation marks as the parameter.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

The **.netuse** command behaves like the **net use** Microsoft MS-DOS command.

If you use **.netuse** during a remote debugging session, this command affects the debugging server, not the debugging client.

The following example shows this command.

```
0:000> .netuse "m:" "\\myserver\myshare" "" ""
```

.noshell (Prohibit Shell Commands)

12/5/2018 • 2 minutes to read • Edit Online

The **.noshell** command prevents you from using **.shell** commands.

```
.noshell
```

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about the command shell and for other ways to disable shell commands, see Using Shell Commands.

Remarks

If you use the **.noshell** command, you cannot use **.shell (Command Shell)** commands as long as the debugger is running, even if you start a new debugging session.

If you are performing remote debugging, this command is useful for security purposes.

.noversion (Disable Version Checking)

12/5/2018 • 2 minutes to read • Edit Online

The **.noversion** command disables all version checking of extension DLLs.

```
.noversion
```

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

The build number of extension DLLs should match the build number of the computer that you are debugging, because the DLLs are compiled and linked with dependencies on specific versions of data structures. If the versions do not match, you typically receive the following message.

```
*** Extension DLL(1367 Free) does not match target system(1552 Free)
```

.ocommand (Expect Commands from Target)

6/25/2019 • 2 minutes to read • Edit Online

The **.ocommand** command enables the target application to send commands to the debugger.

```
.ocommand  String
.ocommand -d
.ocommand
```

Parameters

String
Specifies the command prefix string. *String* can include spaces, but you cannot use C-style control characters such as **\"** and **\n**. You can also enclose *String* in quotation marks. However, if *String* includes a semicolon, leading spaces, or trailing spaces, you must enclose *String* in quotation marks.

-d
Deletes the command prefix string.

Environment

Modes	User mode only
Targets	Live debugging only
Platforms	All

Additional Information

For more information about **OutputDebugString** and other user-mode functions that communicate with a debugger, see the Microsoft Windows SDK documentation.

Remarks

If you use the **.ocommand** command without parameters, the debugger displays the current command prefix string. To clear the existing string, use **.ocommand -d**.

When you have set a command prefix string, any target output (such as the contents of an **OutputDebugString** command) is scanned. If this output begins with the command prefix string, the text of the output that follows the prefix string is treated as a debugger command string and is run. When this text is executed, the command string is not displayed.

The target can include an **.echo (Echo Comment)** command in the output string if you want additional messages. Target output that does not begin with the prefix string is displayed in the typical manner.

After the commands within the command string have been executed, the target remains broken into the debugger, unless the final command is **g (Go)**.

The comparison between the command prefix string and the target output is not case sensitive. (However, subsequent uses of **.ocommand** display the string that you entered with the case preserved).

For this example, assume that you enter the following command in the debugger.

```
0:000> .ocommand magiccommand
```

Then, the target application executes the following line.

```
OutputDebugString("MagicCommand kb;g");
```

The debugger recognizes the command string prefix and executes **kb;g** immediately.

However, the following line does not cause any commands to be executed.

```
OutputDebugString("Command on next line.\nmagiccommand kb;g");
```

There are no commands executed from the preceding example because the command string prefix is not at the beginning of the output, even though it does begin a new line.

Note You should choose a command string prefix that will not likely appear in any target output other than your own commands.

.nvload (NatVis Load)

12/5/2018 • 2 minutes to read • Edit Online

The .nvload command loads a NatVis file into the debugger environment. After the visualization is loaded, it will be used to render data defined in the visualization.

```
.nvload FileName|ModuleName
```

Parameters

FileName | *ModuleName*
Specifies the NatVis file name or module name to load.

The **FileName** is the explicit name of a .natvis file to load. A fully qualified path can be used.

The **ModuleName** is the name of a module in the target process being debugged. All NatVis files which are embedded within the symbol file (PDB) of the named module name are loaded, if there are any available.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information, see Writing debugger type visualizers for C++ using .natvis files.

See also

dx (Display NatVis Expression)

.nvlist (NatVis List)

12/5/2018 • 2 minutes to read • Edit Online

The .nvllist command lists the NatVis files loaded into the debugger environment.

```
.nvlist
```

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information, see Writing debugger type visualizers for C++ using .natvis files.

See also

dx (Display NatVis Expression)

.nvunload (NatVis Unload)

12/5/2018 • 2 minutes to read • Edit Online

The .nvunload command unloads a NatVis file from the debugger environment.

```
.nvunload FileName|ModuleName
```

FileName | *ModuleName*
Specifies the NatVis file name or module name to unload.

The **FileName** is the explicit name of a .natvis file to unload. A fully qualified path can be used.

The **ModuleName** is the name of a module in the target process being debugged. All NatVis files which are embedded within the symbol file (PDB) of the named module name are unloaded.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information, see Writing debugger type visualizers for C++ using .natvis files.

See also

dx (Display NatVis Expression)

.nvunloadall (NatVis Unload All)

12/5/2018 • 2 minutes to read • Edit Online

The .nvunloadall command unloads all of the NatVis files from the debugger environment.

```
.nvunloadall
```

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information, see Writing debugger type visualizers for C++ using .natvis files.

See also

dx (Display NatVis Expression)

.ofilter (Filter Target Output)

6/25/2019 • 2 minutes to read • Edit Online

The **.ofilter** command filters the output from the target application or target computer.

```
.ofilter [/!] String
.ofilter ""
.ofilter
```

Parameters

/!
Reverses the filter so that the debugger displays only output that does not contain *String*. If you do not use this parameter, the debugger displays only output that contains *String*.

String
Specifies the string to match in the target's output. *String* can include spaces, but you cannot use C-style control characters such as **\"** and **\n**. *String* might contain a variety of wildcard characters and specifiers. For more information about the syntax, see String Wildcard Syntax.

You can enclose *String* in quotation marks. However, if *String* includes a semicolon, leading spaces, or trailing spaces, you must use quotation marks. Alphanumeric characters in *String* are converted to uppercase letters, but the actual pattern matching is case insensitive.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about **OutputDebugString** and other user-mode routines, see the Microsoft Windows SDK documentation. For more information about **DbgPrint**, **DbgPrintEx**, and other kernel-mode routines, see the Windows Driver Kit (WDK).

Remarks

If you use the **.ofilter** command without parameters, the debugger displays the current pattern-matching criteria.

To clear the existing filter, use **.ofilter ""**. This command filters any data that is sent by user-mode routines (such as **OutputDebugString**) and kernel-mode routines (such as **DbgPrint**). However, the debugger always displays prompts that **DbgPrompt** sends.

The **DbgPrintEx** and **KdPrintEx** routines supply another method of filtering debugging messages that you do not want.

.open (Open Source File)

12/5/2018 • 2 minutes to read • Edit Online

The **.open** command searches the source path for a source file and opens this file.

```
.open [-m Address] FileName
.open -a Address
```

Parameters

FileName
Specifies the source file name. This name can include an absolute or relative path. Unless you specify an absolute path, the path is interpreted as relative to a directory in the source path.

-m **** *Address*
Specifies an address within the source file. This address must be contained in a known module. You should use the **-m** **** *Address* parameter if the file that *FileName* specifies is not unique. For more information about the syntax, see Address and Address Range Syntax.

The **-m** parameter is required if you are using a source server to retrieve the source files.

-a *Address*
Specifies an address within the source file. This address must be contained in a known module. If the debugger can find the source file, the debugger loads and opens the file, and the line that corresponds to the specified address is highlighted. If the debugger cannot find the source file, the address is displayed in the Disassembly window. For more information about the syntax, see Address and Address Range Syntax.

Environment

You can use the **.open** command only in WinDbg, and you cannot use it in script files.

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about source files and source paths and for other ways to load source files, see Source Path.

.opendump (Open Dump File)

12/5/2018 • 2 minutes to read • Edit Online

The **.opendump** command opens a dump file for debugging.

```
.opendump DumpFile
.opendump /c "DumpFileInArchive" [CabFile]
```

Parameters

DumpFile
Specifies the name of the dump file to open. *DumpFile* should include the file name extension (typically .dmp or .mdmp) and can include an absolute or relative path. Relative paths are relative to the directory that you started the debugger in.

/c "*DumpFileInArchive***"**
Specifies the name of a dump file to debug. This dump file must be contained in the archive file that *CabFile* specifies. You must enclose the *DumpFileInArchive* file in quotation marks.

CabFile
Specifies the name of an archive file to open. *CabFile*should include the file name extension (typically .cab) and can include an absolute or relative path. Relative paths are relative to the directory that you started the debugger in. If you use the **/c** switch to specify a dump file in an archive but you omit *CabFile*, the debugger reuses the archive file that you most recently opened.

Environment

Modes	User mode, kernel mode
Targets	Crash dump only (but you can use this command if other sessions are running)
Platforms	All

Remarks

After you use the **.opendump** command, you must use the **g (Go)** command to finish loading the dump file.

When you are opening an archive file (such as a CAB file), you should use the **/c** switch. If you do not use this switch and you specify an archive for *DumpFile*, the debugger opens the first file that has an .mdmp or .dmp file name extension within this archive.

You can use **.opendump** even if a debugging session is already in progress. This feature enables you to debug more than one crash dump at the same time. For more information about how to control a multiple-target session, see Debugging Multiple Targets.

Note There are complications, when you debug live targets and dump targets together, because commands behave differently for each type of debugging. For example, if you use the **g (Go)** command when the current system is a dump file, the debugger begins executing, but you cannot break back into the debugger, because

the break command is not recognized as valid for dump file debugging.

.outmask (Control Output Mask)

12/5/2018 • 2 minutes to read • Edit Online

The **.outmask** command controls the current output mask.

```
.outmask[-] [/l] Expression
.outmask /a
.outmask /d
```

Parameters

Expression
Specifies the flags to add to the mask. *Expression* can be any ULONG value that specifies the flag bits that you want. For a list of the possible flags, see the table in the Remarks section.

-
Removes the bits that *Expression* specifies from the mask, instead of adding them to the mask.

/l
Preserves the current value of the log file's output mask. If you do not include **/l**, the log file's output mask is the same as the regular output mask.

/a
Activates all mask flags. This parameter is equivalent to **.outmask 0xFFFFFFFF**.

/d
Restores the output mask to the default value. This parameter is equivalent to **.outmask 0x3F7**.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

Each output mask flag enables the debugger to display certain output in the Debugger Command Window. If all of the mask flags are set, all output is displayed.

You should remove output mask flags with caution, because you might be unable to read debugger output.

The following flag values exist.

VALUE	DEFAULT SETTING	DESCRIPTION
1	On	Normal output

VALUE	DEFAULT SETTING	DESCRIPTION
2	On	Error output
4	On	Warnings
8	Off	Additional output
0x10	On	Prompt output
0x20	On	Register dump before prompt
0x40	On	Warnings that are specific to extension operation
0x80	On	Debug output from the target (for example, **OutputDebugString** or **DbgPrint**)
0x100	On	Debug input expected by the target (for example, **DbgPrompt**)
0x200	On	Symbol messages (for example, **!sym noisy**)

.pagein (Page In Memory)

12/5/2018 • 2 minutes to read • Edit Online

The **.pagein** command pages in the specified region of memory.

```
.pagein [Options] Address
```

Parameters

Options
Any of the following options:

/p **** *Process*
Specifies the address of the process that owns the memory that you want to page in. (More precisely, this parameter specifies the address of the EPROCESS block for the process.) If you omit *Process* or specify zero, the debugger uses the current process setting. For more information about the process setting, see **.process (Set Process Context)**

/f
Forces the memory to be paged in, even if the address is in kernel memory and the version of the Microsoft Windows operating system does not support this action.

Address
Specifies the address to page in.

Environment

Modes	Kernel mode only (but not during local kernel debugging)
Targets	Live debugging only
Platforms	All

Remarks

After you run the **.pagein** command, you must use the **g (Go)** command to resume program execution. After a brief time, the target computer automatically breaks into the debugger again.

At this point, the address that you specify is paged in. If you use the **/p** option, the process context is also set to the specified process, exactly as if you used the **.process /i Process** command.

If the address is already paged in, the **.pagein** command still checks that the address is paged in and then breaks back into the debugger. If the address is invalid, this command only breaks back into the debugger.

In Windows Server 2003 and Windows XP, you can page in only user-mode addresses by using **.pagein**. You can override this restriction by using the **/f** switch, but we do not recommend that you use this switch. In Windows Vista, you can safely page in user-mode and kernel-mode memory.

Warning If you use **.pagein** on an address in a kernel stack in Windows Server 2003 or Windows XP, a bug

check might occur.

Requirements

Version	Supported in Windows XP and later versions of Windows.

.pcmd (Set Prompt Command)

12/5/2018 • 2 minutes to read • Edit Online

The **.pcmd** command causes the debugger to issue a command whenever the target stops executing and to display a prompt in the Debugger Command window with register or target state information.

```
.pcmd -s CommandString
.pcmd -c
.pcmd
```

Parameters

-s **** *CommandString*
Specifies a new prompt command string. Whenever the target stops executing, the debugger issues and immediately runs the *CommandString* command. If *CommandString* contains spaces or semicolons, you must enclose it in quotation marks.

-c
Deletes any existing prompt command string.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about the Debugger Command window prompt, see Using Debugger Commands.

Remarks

If you use the **.pcmd** command without parameters, the current prompt command is displayed.

When you set a prompt command by using **.pcmd -s**, the specified *CommandString* is issued whenever the target stops executing (for example, when a **g**, **p**, or **t** command ends). The *CommandString* command is not issued when you use a non-execution command, unless that command displays registers or target state information.

In the following example, the first use of **.pcmd** sets a fixed string that appears with the prompt. The second use of **.pcmd** causes the debugger to display the target's current process ID and thread ID every time that the prompt appears. The special prompt does not appear after the **.ttime** command is used, because that command does not involve execution.

```
0:000> .pcmd
No per-prompt command

0:000> .pcmd -s ".echo Execution is done."
Per-prompt command is '.echo Execution is done.'

0:000> t
Prymes!isPrime+0xd0:
004016c0 837dc400      cmp dword ptr [ebp-0x3c],0x0 ss:0023:0012fe70=00000002
Execution is done.

0:000> t
Prymes!isPrime+0xd4:
004016c4 7507              jnz     Prymes!isPrime+0xdd (004016cd)
 [br=1]
Execution is done.

0:000> .ttime
Created: Thu Aug 21 13:18:59 2003
Kernel:  0 days 0:00:00.031
User:    0 days 0:00:00.000

0:000> .pcmd -s "r $tpid, $tid"
Per-prompt command is 'r $tpid, $tid'

0:000> t
Prymes!isPrime+0xdd:
004016cd ebc0              jmp     Prymes!isPrime+0x9f (0040168f)
$tpid=0000080c $tid=00000514

0:000> t
Prymes!isPrime+0x9f:
0040168f 8b55fc            mov     edx,[ebp-0x4]     ss:0023:0012fea8=00000005
$tpid=0000080c $tid=00000514
```

.pop (Restore Debugger State)

12/5/2018 • 2 minutes to read • Edit Online

The **.pop** command restores the state of the debugger to a state that has previously been saved by using the **.push (Save Debugger State)** command.

```
.pop
.pop /r
.pop /r /q
```

Parameters

/r
Specifies that the saved values of the pseudo-registers $t0 to $t19 should be restored. If **/r** is not included, these values are not affected by the **.pop** command.

/q
Specifies that the command executes quietly. That is, the command executes without displaying any output.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

This command is most useful when used with scripts and debugger command programs so that they can work with one fixed state. If the command is successful, no output is displayed.

.prefer_dml (Prefer Debugger Markup Language)

12/5/2018 • 2 minutes to read • Edit Online

The **.prefer_dml** command sets the default behavior for commands that are capable of providing output in the Debugger Markup Language (DML) format.

```
.prefer_dml 0
.prefer_dml 1
```

Parameters

0

By default, all commands will provide plain text output.

1

By default, commands that are capable of providing DML output will provide DML output.

See also

Debugger Markup Language Commands

.process (Set Process Context)

1/3/2019 • 4 minutes to read • Edit Online

The **.process** command specifies which process is used for the process context.

```
.process [/i] [/p [/r]] [/P] [Process]
```

Parameters

/i
Live debugging only; not during local kernel debugging) Specifies that *Process* is to be debugged *invasively*. This kind of debugging means that the operating system of the target computer actually makes the specified process active. (Without this option, the **.process** command alters the debugger's output but does not affect the target computer itself.) If you use **/i**, you must use the **g (Go)** command to execute the target. After several seconds, the target breaks back in to the debugger, and the specified *Process* is active and used for the process context.

/p
Translates all transition page table entries (PTEs) for this process to physical addresses before access, if you use **/p** and *Process* is nonzero. This translation might cause slowdowns, because the debugger must find the physical addresses for all of the memory that this process uses. Also, the debugger might have to transfer a significant amount of data across the debug cable. (This behavior is the same as **.cache forcedecodeuser**.)

If you include the **/p** option and *Process* is zero or you omit it, the translation is disabled. (This behavior is the same as **.cache noforcedecodeptes**.)

/r
Reloads user-mode symbols after the process context has been set, if you use the **/r** and **/p** options. (This behavior is the same as **.reload /user**.)

/P
(Live debugging and complete memory dumps only) Translates all transition page table entries (PTEs) to physical addresses before access, if you use **/P** and *Process* is nonzero. Unlike the **/p** option, the **/P** option translates the PTEs for all user-mode and kernel-mode processes, not only the specified process. This translation might cause slowdowns, because the debugger must find the physical addresses for all memory in use. Also, the debugger might have to transfer lots of data across the debug cable. (This behavior is the same as **.cache forcedecodeptes**.)

Process
Specifies the address of the process that you want. (More precisely, this parameter specifies the address of the EPROCESS block for this process). The process context is set to this process. If you omit *Process* or specify zero, the process context is reset to the default process for the current system state. (If you used the **/i** option to set process context, you must use the **/i** option to reset the process context.)

Environment

Modes	Kernel mode only
Targets	Live, crash dump

Platforms	All

Additional Information

For more information about the process context and other context settings, see Changing Contexts.

Remarks

Typically, when you are doing kernel debugging, the only visible user-mode address space is the one that is associated with the current process.

The **.process** command instructs the kernel debugger to use a specific user-mode process as the *process context*. This usage has several effects, but the most important is that the debugger has access to the virtual address space of this process. The debugger uses the page tables for this process to interpret all user-mode memory addresses, so you can read and write to this memory.

The **.context (Set User-Mode Address Context)** command has a similar effect. However, the **.context** command sets the *user-mode address context* to a specific page directory, while the **.process** command sets the process context to a specific process. On an x86-based processor, **.context** and **.process** have almost the same effect. However, on an Itanium-based processor, a single process might have more than one page directory. In this situation, the **.process** command is more powerful, because it enables access to all of the page directories that are associated with a process. For more information about the process context, see Process Context.

Note If you are performing live debugging, you should use the **/i** or the **/p** parameter. Without one of these parameters, you cannot correctly display user-mode or session memory.

The **/i** parameter activates the target process. When you use this option, you must execute the target once for this command to take effect. If you execute again, the process context is lost.

The **/p** parameter enables the **forcedecodeuser** setting. (You do not have to use **/p** if the **forcedecodeuser** option is already active.) The process context and the **forcedecodeuser** state remain only until the target executes again.

If you are performing crash dump debugging, the **/i** and **/p** options are not available. However, you cannot access any part of the user-mode process' virtual address space that were paged to disk when the crash occurred.

If you want to use the kernel debugger to set breakpoints in user space, use the **/i** option to switch the target to the correct process context.

The following example shows how to use the **!process** extension to find the address of the EPROCESS block for the desired process.

```
kd> !process 0 0
**** NT ACTIVE PROCESS DUMP ****
PROCESS fe5039e0  SessionId: 0  Cid: 0008    Peb: 00000000  ParentCid: 0000
    DirBase: 00030000  ObjectTable: fe529b68  TableSize:  50.
    Image: System

.....

PROCESS fe3c0d60  SessionId: 0  Cid: 0208    Peb: 7ffdf000  ParentCid: 00d4
    DirBase: 0011f000  ObjectTable: fe3d0f48  TableSize:  30.
    Image: regsvc.exe
```

Now the example uses the **.process** command with this process address.

```
kd> .process fe3c0d60
Implicit process is now fe3c0d60
```

Notice that this command makes the **.context** command unnecessary. The user-mode address context already has the desired value.

```
kd> .context
User-mode page directory base is 11f000
```

This value enables you to examine the address space in various ways. For example, the following example shows the output of the **!peb** extension.

```
kd> !peb
PEB at 7FFDF000
    InheritedAddressSpace:    No
    ReadImageFileExecOptions: No
    BeingDebugged:            No
    ImageBaseAddress:         01000000
    Ldr.Initialized: Yes
    Ldr.InInitializationOrderModuleList: 71f40 . 77f68
    Ldr.InLoadOrderModuleList: 71ec0 . 77f58
    Ldr.InMemoryOrderModuleList: 71ec8 . 77f60
        01000000 C:\WINNT\system32\regsvc.exe
        77F80000 C:\WINNT\System32\ntdll.dll
        77DB0000 C:\WINNT\system32\ADVAPI32.dll
        77E80000 C:\WINNT\system32\KERNEL32.DLL
        77D40000 C:\WINNT\system32\RPCRT4.DLL
        77BE0000 C:\WINNT\system32\secur32.dll
    SubSystemData:     0
    ProcessHeap:       70000
    ProcessParameters: 20000
        WindowTitle:  'C:\WINNT\system32\regsvc.exe'
        ImageFile:    'C:\WINNT\system32\regsvc.exe'
        CommandLine:  'C:\WINNT\system32\regsvc.exe'
        DllPath:
'C:\WINNT\system32;.;C:\WINNT\System32;C:\WINNT\system;C:\WINNT;C:\WINNT\system32;C:\WINNT;C:\WINNT\System32\
Wbem;C:\PROGRA~1\COMMON~1\AUTODE~1'
        Environment:  0x10000
```

.prompt_allow (Control Prompt Display)

12/5/2018 • 2 minutes to read • Edit Online

The **.prompt_allow** command controls what information is displayed during stepping and tracing and whenever the target's execution stops.

```
.prompt_allow {+|-}Item [...]
.prompt_allow
```

Parameters

+
Displays the specified item at the stepping, tracing, and execution prompt. You must add a space before the plus sign (+), but you cannot add a space after it.

-
Prevents the specified item from being displayed at the stepping, tracing, and execution prompt. You must add a space before the minus sign (-), but you cannot add a space after it.

Item
Specifies an item to display or not display. You can specify any number of items. Separate multiple items by spaces. You must add a plus sign (+) or minus sign (-) before each item. You can use the following items:

dis
The disassembled instructions at the current location.

ea
The effective address for the current instruction.

reg
The current state of the most important registers. You can disable register display by using the **pr**, **tr**, or .prompt_allow -reg command. All three of these commands control the same setting, and you can use any of them to override any previous use of these commands.

You can also disable register display by using the l-os command. This setting is separate from the other three commands, as described in the following Remarks section. To control which registers and flags are displayed, use the **rm (Register Mask)** command.

src
The source line that corresponds to the current instruction. You can disable source line display by using the l-ls or .prompt_allow -src; commands. You must enable source line display through both mechanisms to be visible.

sym
The symbol for the current instruction. This symbol includes the current module, function name, and offset.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump

Platforms	All

Additional Information

For more information about commands that affect execution, see Controlling the Target.

Remarks

You can use the **.prompt_allow** command without parameters to show which items are displayed and are not displayed. Every time that you run **.prompt_allow**, the debugger changes the status of only the specified items.

By default, all items are displayed.

If you have used the l+os option, this option overrides any of the **.prompt_allow** options other than **src**.

You can also use a complex command such as the following example.

```
0:000> .prompt_allow -reg -dis +ea
Allow the following information to be displayed at the prompt:
(Other settings can affect whether the information is actually displayed)
   sym - Symbol for current instruction
    ea - Effective address for current instruction
   src - Source info for current instruction
Do not allow the following information to be displayed at the prompt:
   dis - Disassembly of current instruction
   reg - Register state
```

.push (Save Debugger State)

12/5/2018 • 2 minutes to read • Edit Online

The **.push** command saves the current state of the debugger.

```
.push
.push /r
.push /r /q
```

Parameters

/r
Specifies that the current values in the pseudo-registers **$t0** to **$t19** should be saved. If the **/r** parameter is not used, these values are not saved by the **.push** command.

/q
Specifies that the command executes quietly. That is, the command executes without displaying any output.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

This command is most useful when used with scripts and debugger command programs so that they can work with one fixed state. To restore the debugger to a state that was previously saved using this command, use the **.pop (Restore Debugger State)** command. If the command is successful, no output is displayed.

.quit_lock (Prevent Accidental Quit)

12/5/2018 • 2 minutes to read • Edit Online

The **.quit_lock** command sets a password to prevent you from accidentally ending the debugging session.

```
.quit_lock /s NewPassword
.quit_lock /q Password
.quit_lock
```

Parameters

/s **** *NewPassword*
Prevents the debugging session from ending and stores *NewPassword*. You cannot end the debugger session until you use the **.quit_lock /q** command together with this same password. *NewPassword* can be any string. If it contains spaces, you must enclose *NewPassword* in quotation marks.

/q **** *Password*
Enables the debugging session to end. *Password* must match the password that you set with the **.quit_lock /s** command.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

Without parameters, **.quit_lock** displays the current lock status, including the full text of the password.

You can repeat the **.quit_lock /s** command to change an existing password.

When you use **.quit_lock /q**, the lock is removed. This command does not close the debugger. Instead, the command only enables you to exit the session in the typical manner when you want to.

Note The password is not "secret". Any remote user who is attached to the debugging session can use **.quit_lock** to determine the password. The purpose of this command is to prevent accidental use of the **q (Quit)** command. This command is especially useful if restarting the debugging session might be difficult (for example, during remote debugging).

You cannot use the **.quit_lock /s** command in Secure Mode. If you use this command before Secure Mode is activated, the password protection remains, but you cannot change or remove the password.

.readmem (Read Memory from File)

12/5/2018 • 2 minutes to read • Edit Online

The **.readmem** command reads raw binary data from the specified file and copies the data to the target computer's memory.

```
.readmem FileName Range
```

Parameters

FileName
Specifies the name of the file to read. You can specify a full path or only the file name. If the file name contains spaces, enclose *FileName* in quotation marks. If you do not specify a path, the debugger uses the current directory.

Range
Specifies the address range for putting the data in memory. This parameter can contain a starting and ending address or a starting address and an object count. For more information about the syntax, see Address and Address Range Syntax.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

The memory data is copied literally to the target computer. The debugger does not parse the data in any way. For example, the **.readmem myfile 1000 10** command copies 10 bytes from the Myfile file and stores them in the target computer's memory, starting at address 1000.

The **.readmem** command is the opposite of the **.writemem (Write Memory to File)** command.

.reboot (Reboot Target Computer)

12/5/2018 • 2 minutes to read • Edit Online

The **.reboot** command restarts the target computer.

```
.reboot
```

Environment

Modes	Kernel mode only
Targets	Live debugging only
Platforms	All

Additional Information

For more information about related commands and an explanation of how the restart process affects the debugger, see Crashing and Rebooting the Target Computer.

.record_branches (Enable Branch Recording)

12/5/2018 • 2 minutes to read • Edit Online

The **.record_branches** command enables the recording of branches that the target's code executed.

```
.record_branches {1|0}
.record_branches
```

Environment

Modes	User mode, kernel mode
Targets	Live debugging only
Platforms	x64-based only

Remarks

The **.record_branches 1** command enables the recording of branches in the target's code. The **.record_branches 0** command disables this recording.

Without parameters, **.record_branches** displays the current status of this setting.

.reload (Reload Module)

8/27/2019 • 4 minutes to read • Edit Online

The **.reload** command deletes all symbol information for the specified module and reloads these symbols as needed. In some cases, this command also reloads or unloads the module itself.

```
.reload [Options] [Module[=Address[,Size[,Timestamp]]]]
.reload -?
```

Parameters

Options
Any of the following options:

/d
Reload all modules in the debugger's module list. (When you omit all parameters, this situation is the default during user-mode debugging.)

/f
Forces the debugger to immediately load the symbols. This parameter overrides *lazy symbol loading*. For more information, see the following Remarks section.

/i
Ignores a mismatch in the .pdb file versions. (If you do not include this parameter, the debugger does not load mismatched symbol files.) When you use **/i**, **/f** is used also, even if you do not explicitly specify it.

/l
Lists the modules but does not reload their symbols. (In kernel mode, this parameter provides output similar to the **lm** command.)

/n
Reloads kernel symbols only. This parameter does not reload any user symbols. (You can use this option only during kernel-mode debugging.)

/o
Forces the cached files in a symbol server's downstream store to be overwritten. When you use this flag, you should also include **/f**. By default, the downstream store files are never overwritten.

Because the symbol server uses distinct file names for the symbols of every different build of a binary, you do not have to use this option unless you believe your downstream store has become corrupted.

/s
Reloads all modules in the system's module image list. (When you omit all parameters, this situation is the default during kernel-mode debugging.)

If you are loading an individual system module by name while you perform user-mode debugging, you must include **/s**.

/u
Unloads the specified module and all its symbols. The debugger unloads any loaded module whose name matches *Module*, regardless of the full path. Image names are also searched. For more information, see the note in the following Remarks section.

/unl
Reloads symbols based on the image information in the unloaded module list.

/user
Reloads user symbols only. (You can use this option only during kernel-mode debugging.)

/v
Turns on verbose mode.

/w
Treats *Module* as a literal string. This treatment prevents the debugger from expanding wildcard characters.

Module
Specifies the name of an image on the target system for which to reload symbols on the host computer. *Module* should include the name and file name extension of the file. Unless you use the **/w** option, *Module* might contain a variety of wildcard characters and specifiers. For more information about the syntax, see String Wildcard Syntax. If you omit *Module*, the behavior of the **.reload** command depends on which *Options* you use.

Address
Specifies the base address of the module. Typically, you have to have this address only if the image header has been corrupted or is paged out.

Size
Specifies the size of the module image. In many situations, the debugger knows the correct size of the module. When the debugger does not know the correct size, you should specify *Size*. This size can be the actual module size or a larger number, but the size should not be a smaller number. Typically, you have to have this size only if the image header has been corrupted or is paged out.

Timestamp
Specifies the timestamp of the module image. In many situations, the debugger knows the correct timestamp of the module. When the debugger does not know the timestamps, you should specify *Timestamp*. Typically, you have to have this timestamp only if the image header has been corrupted or is paged out.

Note There must be no blank space between the *Address*, *Size*, and *Timestamp* parameters.

-?
Displays a short help text for this command.

Environment

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about deferred (lazy) symbol loading, see Deferred Symbol Loading. For more information about other symbol options, see Setting Symbol Options.

Remarks

The **.reload** command does not cause symbol information to be read. Instead, this command lets the debugger know that the symbol files might have changed or that a new module should be added to the

module list. This command causes the debugger to revise its module list and delete its symbol information for the specified modules. The actual symbol information is not read from the individual .pdb files until the information is needed. (This kind of loading is known as *lazy symbol loading* or *deferred symbol loading*.)

You can force symbol loading to occur by using the **/f** option or by issuing an **ld (Load Symbols)** command.

The **.reload** command is useful if the system stops responding (that is, crashes), which might cause you to lose symbols for the target computer that is being debugged. The command can also be useful if you have updated the symbol tree.

If the image header is incorrect for some reason, such as the module being unloaded, or is paged out, you can load symbols correctly by using the **/unl** argument, or specifying both *Address* and *Size*.

The **.reload /u** command performs a broad search. The debugger first tries to match *Module* with an exact module name, regardless of path. If the debugger cannot find this match, *Module* is treated as the name of the loaded image. For example, if the HAL that resides in memory has the module name of halacpi.dll, both of the following commands unload its symbols.

```
kd> .reload /u halacpi.dll

kd> .reload /u hal
```

If you are performing user-mode debugging and want to load a module that is not part of the target application's module list, you must include the **/s** option, as the following example shows.

```
0:000> .reload /u ntdll.dll
Unloaded ntdll.dll

0:000> .reload /s /f ntdll.dll
```

.remote (Create Remote.exe Server)

12/5/2018 • 2 minutes to read • Edit Online

The **.remote** command starts a Remote.exe Server, enabling a remote connection to the current debugging session.

```
.remote session
```

Parameters

session
Specifies a name that you give to the debugging session.

Environment

You can use the **.remote** command in KD and CDB, but you cannot use it in WinDbg.

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about how to use Remote.exe Servers and Remote.exe Clients, see Remote Debugging Through Remote.exe.

Remarks

The **.remote** command creates a Remote.exe process and turns the current debugger into a Remote.exe Server. This server enables a Remote.exe Client to connect to the current debugging session.

.remote_exit (Exit Debugging Client)

12/5/2018 • 2 minutes to read • Edit Online

The **.remote_exit** command exits the debugging client but does not end the debugging session.

```
.remote_exit [FinalCommands]
```

Parameters

FinalCommands
Specifies a command string to pass to the debugging server. You should separate multiple commands by using semicolons. These commands are passed to the debugging server and the connection is then broken.

Environment

You can use the **.remote_exit** command only in a script file. You can use it in KD and CDB, but you cannot use it in WinDbg.

Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about script files, see Using Script Files. For more information about debugging clients and debugging servers, see Remote Debugging Through the Debugger.

Remarks

If you are using KD or CDB directly, instead of using a script, you can exit from the debugging client by using the **CTRL+B** key.

You cannot exit from a debugging client through a script that is executed in WinDbg.

.restart (Restart Target Application)

12/5/2018 • 2 minutes to read • Edit Online

The **.restart** command restarts the target application.

Do not confuse this command with the **.restart (Restart Kernel Connection)** command, which works only in kernel mode.

```
.restart
```

Environment

Modes	User mode only
Targets	Live debugging only
Platforms	All

Additional Information

For more information about how to issue this command and an overview of related commands, see Controlling the Target.

Remarks

CDB and WinDbg can restart a target application if the debugger originally created the application. You can use the **.restart** command even if the target application has already closed.

However, if the application is running and the debugger is later attached to the process, the **.restart** command has no effect.

After the process is restarted, it immediately breaks into the debugger.

In WinDbg, use the View | WinDbg Command Line command if you started your target from the WinDbg command prompt and you want to review this command line before you decide whether to use **.restart**.

.restart (Restart Kernel Connection)

12/5/2018 • 2 minutes to read • Edit Online

The **.restart** command restarts the kernel connection.

Do not confuse this command with the **.restart (Restart Target Application)** command, which works only in user mode.

```
.restart
```

Environment

You can use the **.restart** command only in KD.

Modes	Kernel mode only
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about reestablishing contact with the target, see Synchronizing with the Target Computer.

Remarks

The **.restart** command is similar to the **CTRL+R (Re-synchronize)** command, except that **.restart** is even more extensive in its effect. This command is equivalent to ending the debugger and then attaching a new debugger to the target computer.

The **.restart** command is most useful when you are performing remote debugging through remote.exe and ending and restarting the debugger might be difficult. However, you cannot use **.restart** from a debugging client if you are performing remote debugging through the debugger.

.rrestart (Register for Restart)

12/5/2018 • 2 minutes to read • Edit Online

The **.rrestart** command registers the debugging session for restart in case of a reboot or an application failure.

```
.rrestart
```

Remarks

This command does not work for elevated debugger sessions.

See also

.urestart

.scroll_prefs (Control Source Scrolling Preferences)

12/5/2018 • 2 minutes to read • Edit Online

The **.scroll_prefs** command controls the positioning of the source in a Source window when scrolling to a line.

```
.scroll_prefs Before After
.scroll_prefs
```

Parameters

Before
Specifies how many source lines before the line you are scrolling to should be visible in the Source window.

After
Specifies how many source lines after the line you are scrolling to should be visible in the Source window.

Environment

This command is available only in WinDbg and cannot be used in script files.

Modes	user mode, kernel mode
Targets	live, crash dump
Platforms	all

Remarks

When this command is used with no parameters, the current source scrolling preferences are displayed.

.scriptdebug (Debug JavaScript)

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

Use the **.scriptdebug** command to debug JavaScript scripts.

```
.scriptdebug FileName
```

Parameters

FileName

Specifies the name of the debugger JavaScript script to debug.

Environment

Modes	all
Targets	all
Platforms	all

Additional Information

For an overview of JavaScript debugging, see JavaScript Debugger Scripting - JavaScript Debugging.

> **NOTE**
> To use JavaScript Debugging with WinDbg Preview, run the debugger as Administrator.

Remarks

Before you debug a JavaScript completed the following steps.

1. Load the JavaScript scripting provider using the **.load (Load Extension DLL)** command.

   ```
   0:000> .load jsprovider.dll
   ```

2. Load the sample script.

   ```
   0:000> .scriptload C:\MyScripts\DebuggableSample.js
   ```

To start actively debugging the script use the **.scriptdebug** command.

```
0:000> .scriptdebug C:\MyScripts\DebuggableSample.js
>>> ****** DEBUGGER ENTRY DebuggableSample ******
           No active debug event!

>>> Debug [DebuggableSample <No Position>] >
```

Once you see the prompt >>> *Debug [DebuggableSample]* > and a request for input, you are inside the script debugger.

Use the **.help** command or **?** to display a list of commands in the JavaScript debugging environment.

```
>>> Debug [DebuggableSample <No Position>] >.help
Script Debugger Commands (*NOTE* IDs are **PER SCRIPT**):
    ? ................................ Get help
    ? <expr>  ........................ Evaluate expression <expr> and display result
    ?? <expr>  ....................... Evaluate expression <expr> and display result
    |  ................................ List available scripts
    |<scriptid>s  ..................... Switch context to the given script
    bc <bpid>  ....................... Clear breakpoint by specified <bpid>
    bd <bpid>  ....................... Disable breakpoint by specified <bpid>
    be <bpid>  ....................... Enable breakpoint by specified <bpid>
    bl  .............................. List breakpoints
    bp <line>:<column>  .............. Set breakpoint at the specified line and column
    bp <function-name>  .............. Set breakpoint at the (global) function specified by the given name
    bpc  ............................. Set breakpoint at current location
    dv  .............................. Display local variables of current frame
    g  ............................... Continue script
    gu   ............................. Step out
    k  ............................... Get stack trace
    p  ............................... Step over
    q  ............................... Exit script debugger (resume execution)
    sx  .............................. Display available events/exceptions to break on
    sxe <event>  ..................... Enable break on <event>
    sxd <event>  ..................... Disable break on <event>
    t  ............................... Step in
    .attach <scriptId>  .............. Attach debugger to the script specified by <scriptId>
    .detach [<scriptId>]  ............ Detach debugger from the script specified by <scriptId>
    .frame <index>  .................. Switch to frame number <index>
    .f+  ............................. Switch to next stack frame
    .f-  ............................. Switch to previous stack frame
    .help  ........................... Get help
```

Events

Use the **sx** script debugger command to see the list of events that can be trapped.

```
>>> Debug [DebuggableSample <No Position>] >sx
sx
    ab  [   inactive] .... Break on script abort
    eh  [   inactive] .... Break on any thrown exception
    en  [   inactive] .... Break on entry to the script
    uh  [     active] .... Break on unhandled exception
```

Use the **sxe** script debugger command to enable any of the break behaviors. For example to turn on break on entry so that the script will trap into the script debugger as soon as any code within it executes, use this command.

```
>>> Debug [DebuggableSample <No Position>] >sxe en
sxe en
Event filter 'en' is now active
```

Use the **sxd** script debugger command to disable any of the breakpoint behaviors.

```
>>> Debug [DebuggableSample 34:5] >sxd en
sxd en
Event filter 'en' is now inactive
```

Stack trace

Use the **k** command to display a stack trace.

```
>>> Debug [DebuggableSample 34:5] >k
k
    ##  Function                        Pos    Source Snippet
-> [00] throwAndCatch                   034:05 (var curProc = host.currentProcess)
   [01] outer                           066:05 (var foo = throwAndCatch())
   [02] outermost                       074:05 (var result = outer())
```

Enumerating variables

Use **??** to enumerate the values of JavaScript variables.

```
>>> Debug [DebuggableSample 34:5] >??someObj
??someObj
someObj          : {...}
    __proto__        : {...}
    a                : 0x63
    b                : {...}
```

Breakpoints

Use the following breakpoint commands to work with additional breakpoints.

bp	Set a breakpoint
bd	Disable the breakpoint
be	Enable the breakpoint
bc	Clear the breakpoint
bpc	Set breakpoint on current line
bl	List the breakpoint(s)

Flow control - navigation

Use the following commands to move forward in the script.

p	Step over
t	Step in
g	Continue script
gu	Step out

Frames

Use the following commands to work with frames.

.frame	Switch to frame number
.f+	Switch to next stack frame
.f+	Switch to previous stack frame

Quiting

Use the **.detach** command to detach the JavaScript debugger.

```
>>> Debug [DebuggableSample 34:5] >.detach
.detach
Debugger has been detached from script!
```

Use the **q** command to quit the JavaScript debugger.

```
>>> Debug [<NONE> ] >q
q
```

.scriptlist (List Loaded Scripts)

12/5/2018 • 2 minutes to read • Edit Online

The **.scriptlist** command lists the loaded scripts.

```
.scriptlist
```

Parameters

None

Environment

Modes	user mode, kernel mode
Targets	live, crash dump
Platforms	all

Additional Information

The .scriptlist command will list any scripts which have been loaded via the .scriptload command.

If the TestScript was successfully loaded using .scriptload, the .scriptlist command would display the name of the loaded script.

```
0:000> .scriptlist
Command Loaded Scripts:
    JavaScript script from 'C:\WinDbg\Scripts\TestScript.js'
```

Requirements

Before using any of the .script commands, a scripting provider needs to be loaded. Use the **.load (Load Extension DLL)** command to load the JavaScript provider.

```
0:000> .load jsprovider.dll
```

See also

JavaScript Debugger Scripting

.scriptload (Load Script)

.scriptload (Load Script)

12/5/2018 • 2 minutes to read • Edit Online

The **.scriptload** command will load and execute the specified script file.

```
.scriptload ScriptFile
```

Parameters

ScriptFile
Specifies the name of the script file to load. *ScriptFile* should include the .js file name extension. Absolute or relative paths can be used. Relative paths are relative to the directory that you started the debugger in. File paths containing spaces are not supported.

Environment

Modes	user mode, kernel mode
Targets	live, crash dump
Platforms	all

Additional Information

The .scriptload command will load a script and execute a script. The following command shows the successful load of TestScript.js.

```
0:000> .scriptload C:\WinDbg\Scripts\TestScript.js
JavaScript script successfully loaded from 'C:\WinDbg\Scripts\TestScript.js'
```

If there are any errors in the initial load and execution of the script, the errors will be displayed to console, including the line number and error message.

```
0:000:x86> .scriptload C:\WinDbg\Scripts\TestScript.js
0:000> "C:\WinDbg\Scripts\TestScript.js" (line 11 (@ 1)): Error (0x80004005): Syntax error
Error: Unable to execute JavaScript script 'C:\WinDbg\Scripts\TestScript.js'
```

The .scriptload command will execute the following in a JavaScript.

- root code
- intializeScript function (if present in the script)

When a script is loaded using the .scriptload command, the intializeScript function and the root code of the script is executed, the names which are present in the script are bridged into the root namespace of the debugger (dx Debugger) and the script stays resident in memory until it is unloaded and all references to its objects are released.

The script can provide new functions to the debugger's expression evaluator, modify the object model of the

debugger, or can act as a visualizers in much the same way that a NatVis visualizer does. For more information about NavVis and the debugger, see **dx (Display NatVis Expression)**.

For more information about working with JavaScript, see JavaScript Debugger Scripting. For more information about the debugger objects, see Native Objects in JavaScript Extensions.

Requirements

Before using any of the .script commands, a scripting provider needs to be loaded. Use the **.load (Load Extension DLL)** command to load the JavaScript provider.

```
0:000> .load jsprovider.dll
```

See also

.scriptunload (Unload Script)

JavaScript Debugger Scripting

.scriptproviders (List Script Providers)

12/5/2018 • 2 minutes to read • Edit Online

The **.scriptproviders** command lists the active script providers.

```
.scriptproviders
```

Parameters

None

Environment

Modes	user mode, kernel mode
Targets	live, crash dump
Platforms	all

Additional Information

The .scriptproviders command will list all the script languages which are presently understood by the debugger and the extension under which they are registered. Any file ending in ".NatVis" is understood as a NatVis script and any file ending in ".js" is understood as a JavaScript script. Either type of script can be loaded with the .scriptload command.

In the example below, the JavaScript and NatVis providers are loaded.

```
0:000> .scriptproviders
Available Script Providers:
    NatVis (extension '.NatVis')
    JavaScript (extension '.js')
```

Requirements

Before using any of the .script commands, a scripting provider needs to be loaded. Use the **.load (Load Extension DLL)** command to load the JavaScript provider.

```
0:000> .load jsprovider.dll
```

See also

JavaScript Debugger Scripting

.scriptload (Load Script)

.scriptrun (Run Script)

12/5/2018 • 2 minutes to read • Edit Online

The .scriptrun command will load and run a JavaScript.

```
.scriptrun ScriptFile
```

Parameters

ScriptFile
Specifies the name of the script file to load and execute. *ScriptFile* should include the .js file name extension. Absolute or relative paths can be used. Relative paths are relative to the directory that you started the debugger in. File paths containing spaces are not supported.

Environment

Modes	user mode, kernel mode
Targets	live, crash dump
Platforms	all

Additional Information

The .scriptrun command will load a script and, execute the following code.

- root
- intializeScript
- invokeScript

A confirmation message is displayed when the code is loaded and executed.

```
0:000> .scriptrun C:\WinDbg\Scripts\helloWorld.js
JavaScript script successfully loaded from 'C:\WinDbg\Scripts\helloWorld.js'
Hello World!  We are in JavaScript!
```

Any object model manipulations made by the script will stay in place until the script is subsequently unloaded or is run again with different content.

This table summarizes which functions are executed by .scriptload and .scriptrun.

	.scriptload	.scriptrun
root	yes	yes
initializeScript	yes	yes
invokeScript		yes

uninitializeScript		

You can use this code to see which functions are called with the .script run command.

```
// Root of Script
host.diagnostics.debugLog("***>; Code at the very top (root) of the script is always run \n");


function initializeScript()
{
    // Add code here that you want to run every time the script is loaded.
    // We will just send a message to indicate that function was called.
    host.diagnostics.debugLog("***>; initializeScript was called \n");
}

function invokeScript()
{
    // Add code here that you want to run every time the script is executed.
    // We will just send a message to indicate that function was called.
    host.diagnostics.debugLog("***>; invokeScript was called \n");
}
```

For more information about working with JavaScript, see JavaScript Debugger Scripting. For more information about the debugger objects, see Native Objects in JavaScript Extensions.

Requirements

Before using any of the .script commands, a scripting provider needs to be loaded. Use the **.load (Load Extension DLL)** command to load the JavaScript provider dll.

```
0:000> .load C:\ScriptProviders\jsprovider.dll
```

See also

.scriptload (Load Script)

JavaScript Debugger Scripting

.scriptunload (Unload Script)

12/5/2018 • 2 minutes to read • Edit Online

The **.scriptunload** command unloads the specified script.

```
.scriptunload ScriptFile
```

Parameters

ScriptFile
Specifies the name of the script file to unload. *ScriptFile* should include the .js file name extension. Absolute or relative paths can be used. Relative paths are relative to the directory that you started the debugger in. File paths containing spaces are not supported.

Environment

Modes	user mode, kernel mode
Targets	live, crash dump
Platforms	all

Additional Information

The .scriptunload command unloads a loaded script. Use the following command syntax to unload a script

```
0:000:x86> .scriptunload C:\WinDbg\Scripts\TestScript.js
JavaScript script unloaded from 'C:\WinDbg\Scripts\TestScript.js'
```

If there are outstanding references to objects in a script, the contents of the script will be unlinked but the script may remain in memory until all such references are released.

For more information about working with JavaScript, see JavaScript Debugger Scripting. For more information about the debugger objects, see Native Objects in JavaScript Extensions.

Requirements

Before using any of the .script commands, a scripting provider needs to be loaded. Use the **.load (Load Extension DLL)** command to load the JavaScript provider dll.

```
0:000> .load C:\ScriptProviders\jsprovider.dll
```

See also

.scriptload (Load Script)

JavaScript Debugger Scripting

.secure (Activate Secure Mode)

12/5/2018 • 2 minutes to read • Edit Online

The **.secure** command activates or displays the status of Secure Mode.

```
.secure 1
.secure
```

Environment

Secure Mode can only be enabled while the debugger is dormant. Secure Mode applies only to kernel-mode sessions because, by definition, Secure Mode prevents user-mode debugging operations.

Modes	kernel mode only
Targets	live, crash dump
Platforms	all

Additional Information

For details, see Secure Mode.

Remarks

To activate Secure Mode, use the command **.secure 1** (or **.secure** followed by any nonzero value).

The command **.secure** will show whether Secure Mode is currently active.

.send_file (Send File)

12/5/2018 • 2 minutes to read • Edit Online

The **.send_file** command copies files. If you are performing remote debugging through a process server, it sends a file from the smart client's computer to the process server's computer.

```
.send_file [-f] Source Destination
.send_file [-f] -s Destination
```

Parameters

-f

Forces file creation. By default, **.send_file** will not overwrite any existing files. If the -f switch is used, the destination file will always be created, and any existing file with the same name will be overwritten.

Source

Specifies the full path and filename of the file to be sent. If you are debugging through a process server, this file must be located on the computer where the smart client is running.

Destination

Specifies the directory where the file is to be written. If you are debugging through a process server, this directory name is evaluated on the computer where the process server is running.

-s

Causes the debugger to copy all loaded symbol files.

Environment

Modes	user mode, kernel mode
Targets	live, crash dump
Platforms	all

Remarks

This command is particularly useful when you have been performing remote debugging through a process server, but wish to begin debugging locally instead. In this case you can use the .send_file -s command to copy all the symbol files that the debugger has been using to the process server. These symbol files can then be used by a debugger running on the local computer.

.server (Create Debugging Server)

1/3/2019 • 4 minutes to read • Edit Online

The **.server** command starts a debugging server, allowing a remote connection to the current debugging session.

```
.server npipe:pipe=PipeName[,hidden][,password=Password][,IcfEnable]
.server tcp:port=Socket[,hidden][,password=Password][,ipversion=6][,IcfEnable]
.server tcp:port=Socket,clicon=Client[,password=Password][,ipversion=6]
.server com:port=COMPort,baud=BaudRate,channel=COMChannel[,hidden][,password=Password]
.server spipe:proto=Protocol,{certuser=Cert|machuser=Cert},pipe=PipeName[,hidden][,password=Password]
.server ssl:proto=Protocol,{certuser=Cert|machuser=Cert},port=Socket[,hidden][,password=Password]
.server ssl:proto=Protocol,{certuser=Cert|machuser=Cert},port=Socket,clicon=Client[,password=Password]
```

Parameters

PipeName
When NPIPE or SPIPE protocol is used, *PipeName* is a string that will serve as the name of the pipe. Each pipe name should identify a unique debugging server. If you attempt to reuse a pipe name, you will receive an error message. *PipeName* must not contain spaces or quotation marks. *PipeName* can include a numerical **printf**-style format code, such as %x or %d. The debugger will replace this with the process ID of the debugger. A second such code will be replaced with the thread ID of the debugger.

Socket
When TCP or SSL protocol is used, *Socket* is the socket port number.

It is also possible to specify a range of ports separated by a colon. The debugger will check each port in this range to see if it is free. If it finds a free port and no error occurs, the debugging server will be created. The debugging client will have to specify the actual port being used to connect to the server. To determine the actual port, use any of the methods described in **Searching for Debugging Servers**; when this debugging server is displayed, the port will be followed by two numbers separated by a colon. The first number will be the actual port used; the second can be ignored. For example, if the port was specified as port=51:60, and port 53 was actually used, the search results will show "port=53:60". (If you are using the **clicon** parameter to establish a reverse connection, the debugging client can specify a range of ports in this manner, while the server must specify the actual port used.)

clicon=*Client*
When TCP or SSL protocol is used and the **clicon** parameter is specified, a *reverse connection* will be opened. This means that the debugging server will try to connect to the debugging client, instead of letting the client initiate the contact. This can be useful if you have a firewall that is preventing a connection in the usual direction. *Client* specifies the network name of the machine on which the debugging client exists or will be created. The two initial backslashes (\) are optional.

When **clicon** is used, it is best to start the debugging client before the debugging server is created, although the usual order (server before client) is also permitted. A reverse-connection server will not appear when another debugger displays all active servers.

COMPort
When COM protocol is used, *COMPort* specifies the COM port to be used. The prefix COM is optional (for example, both "com2" and "2" are acceptable).

BaudRate
When COM protocol is used, *BaudRate* specifies the baud rate at which the connection will run. Any baud rate

that is supported by the hardware is permitted.

COMChannel
If COM protocol is used, *COMChannel* specifies the COM channel to be used in communicating with the debugging client. This can be any value between 0 and 254, inclusive.

Protocol
If SSL or SPIPE protocol is used, *Protocol* specifies the Secure Channel (S-Channel) protocol. This can be any one of the strings tls1, pct1, ssl2, or ssl3.

Cert
If SSL or SPIPE protocol is used, *Cert* specifies the certificate. This can either be the certificate name or the certificate's thumbprint (the string of hexadecimal digits given by the certificate's snapin). If the syntax **certuser=***Cert* is used, the debugger will look up the certificate in the system store (the default store). If the syntax **machuser=***Cert* is used, the debugger will look up the certificate in the machine store. The specified certificate must support server authentication.

hidden
Prevents the server from appearing when another debugger displays all active servers.

password=*Password*
Requires a debugging client to supply the specified password in order to connect to the debugging session. *Password* can be any alphanumeric string, up to twelve characters in length.

ipversion=6
(Debugging Tools for Windows 6.6.07 and earlier only) Forces the debugger to use IP version 6 rather than version 4 when using TCP to connect to the Internet. In Windows Vista and later versions, the debugger attempts to auto-default to IP version 6, making this option unnecessary.

IcfEnable
Causes the debugger to enable the necessary port connections for TCP or named pipe communication when the Internet Connection Firewall is active. By default, the Internet Connection Firewall disables the ports used by these protocols. When **IcfEnable** is used with a TCP connection, the debugger causes Windows to open the port specified by the Socket parameter. When **IcfEnable** is used with a named pipe connection, the debugger causes Windows to open the ports used for named pipes (ports 139 and 445). The debugger does not close these ports after the connection terminates.

Environment

Modes	user mode, kernel mode
Targets	live, crash dump
Platforms	all

Additional Information

For full details on how to start a debugging server, see **Activating a Debugging Server**. For examples, see Client and Server Examples.

Remarks

This command turns the current debugger into a debugging server. This allows you to start the server after the debugger is already running, whereas the -server command-line option can only be issued when the debugger is started.

This permits a debugging client to connect to the current debugging session. Note that it is possible to start multiple servers using different options, allowing different kinds of debugging clients to join the session.

.servers (List Debugging Servers)

12/5/2018 • 2 minutes to read • Edit Online

The **.servers** command lists all debugging servers that have been established by this debugger.

```
.servers
```

Environment

Modes	user mode, kernel mode
Targets	live, crash dump
Platforms	all

Additional Information

For full details on debugging servers, see Remote Debugging Through the Debugger.

Remarks

The output of the **.servers** command lists all the debugging servers started by the debugger on which this command is issued. The output is formatted so that it can be used literally as the argument for the -remote command-line option or pasted into the WinDbg dialog box.

Each debugging server is identified by a unique ID. This ID can be used as the argument for the **.endsrv (End Debugging Server)** command, if you wish to terminate the debugging server.

The **.servers** command does not list debugging servers started on this computer by different instances of the debugger, nor does it list process servers or KD connection servers.

.setdll (Set Default Extension DLL)

12/5/2018 • 2 minutes to read • Edit Online

The **.setdll** command changes the default extension DLL for the debugger.

```
.setdll DLLName
!DLLName.setdll
```

Parameters

DLLName
The name and path of the extension DLL. If the full path was specified when the DLL was loaded, it needs to be given in full here as well.

Environment

Modes	user mode, kernel mode
Targets	live, crash dump
Platforms	all

Additional Information

For details on loading, unloading, and controlling extensions, see Loading Debugger Extension DLLs. For details on executing extension commands, see Using Debugger Extension Commands.

Remarks

The debugger maintains a default extension DLL that is implicitly loaded when the debugger is started. This allows the user to specify an extension command without first having to load an extension DLL. This command allows modification of which DLL is loaded as the default DLL.

When a command is issued, the debugger looks for it in the default extension first. If a match is not found, all other loaded extension DLLs are searched in the order they were loaded.

.shell (Command Shell)

12/5/2018 • 2 minutes to read • Edit Online

The **.shell** command launches a shell process and redirects its output to the debugger, or to a specified file.

```
.shell [Options] [ShellCommand]
.shell -i InFile [-o OutFile [-e ErrFile]] [Options] ShellCommand
```

Parameters

InFile
Specifies the path and file name of a file to be used for input. If you intend to offer no input after the initial command, you can specify a single hyphen (-) instead of *InFile*, with no space before the hyphen.

OutFile
Specifies the path and file name of a file to be used for standard output. If **-o** **** *OutFile* is omitted, output is sent to the Debugger Command window. If you do not want this output displayed or saved in a file, you can specify a single hyphen (-) instead of *OutFile*, with no space before the hyphen.

ErrFile
Specifies the path and file name of a file to be used for error output. If -e ErrFile is omitted, error output is sent to the same place as standard output. If you do not want this output displayed or saved in a file, you can specify a single hyphen (-) instead of *ErrFile*, with no space before the hyphen.

Options
Can be any number of the following options:

-ci "*Commands***"**
Processes the specified debugger commands, and then passes their output as an input file to the process being launched. *Commands* can be any number of debugger commands, separated by semicolons, and enclosed in quotation marks.

-x
Causes any process being spawned to be completely detached from the debugger. This allows you to create processes which will continue running even after the debugging session ends.

ShellCommand
Specifies the application command line or Microsoft MS-DOS command to be executed.

Environment

Modes	user mode, kernel mode
Targets	live, crash dump
Platforms	all

Additional Information

For other ways of accessing the command shell, see Using Shell Commands.

Remarks

The **.shell** command is not supported when the output of a user-mode debugger is redirected to the kernel debugger. For more information about redirecting output to the kernel debugger (sometimes called NTSD over KD), see Controlling the User-Mode Debugger from the Kernel Debugger.

The entire line after the **.shell** command will be interpreted as a Windows command (even if it contains a semicolon). This line should not be enclosed in quotation marks. There must be a space between **.shell** and the *ShellCommand* (additional leading spaces are ignored).

The output from the command will appear in the Debugger Command window, unless the **-o** **** *OutFile* parameter is used.

Issuing a **.shell** command with no parameters will activate the shell and leave it open. All subsequent commands will be interpreted as Windows commands. During this time, the debugger will display messages reading **<.shell process may need input>**, and the WinDbg prompt will be replaced with an **Input>** prompt. Sometimes, a separate Command Prompt window will appear when the debugger leaves the shell open. This window should be ignored; all input and output will be done through the Debugger Command window.

To close this shell and return to the debugger itself, type **exit** or **.shell_quit**. (The **.shell_quit** command is more powerful, because it works even if the shell is frozen.)

This command cannot be used while debugging CSRSS, because new processes cannot be created without CSRSS being active.

You can use the -ci flag to run one or more debugger commands and then pass their output to a shell process. For example, you could pass the output from the **!process 0 7** command to a Perl script by using the following command:

```
0:000> .shell -ci "!process 0 7" perl.exe parsemyoutput.pl
```

.settings (Set Debug Settings)

12/5/2018 • 2 minutes to read • Edit Online

The **.settings** command sets, modifies, displays, loads and saves settings in the Debugger.Settings namespace.

```
.settings set  namespace.setting=value
.settings set namespace.setting+=value
.settings save [file path]
.settings load file path
.settings list [namespace][-v]
.settings help
```

Parameters

.settings set parameters

namespace.setting=value
Sets or modifies a setting. When specifying file paths use slash escaping, for example C:\\Symbols\\.

Examples:

```
.settings set Display.PreferDMLOutput=false
```

```
.settings set Sources.DisplaySourceLines=true
```

```
.settings set Symbols.Sympath="C:\\Symbols\\"
```

namespace.setting+=value
Specifies that the new value will be appended to (rather than replace) the previous value.

Example:

```
.settings set Extensions.ExtensionSearchPath+=";C:\\MyExtension\\"
```

.setting save parameters

file path
Saves all of the values in the Debugger.Settings namespace to the specified XML file.

none
If a file path is not provided, the settings will be saved to the last file that was loaded or saved to. If a previous file does not exist, a file named config.xml will be created in the directory that the debugger executable was loaded from.

.setting load parameters

file path
Loads all the settings from an XML settings file. Loading settings will change only the settings that are defined in that file. Any previously loaded or changed settings that do not appear in that file will not be modified. This file will be treated as your default save path until the next save or load operation.

.setting list parameters

namespace
List all settings in the given namespace and their values.

-v
The –v flag causes a description of the setting to be displayed.

.setting help parameters

None
Lists all of the settings in the Debugger namespace and their description.

Namespace
Lists all settings in the given namespace and their description.

Environment

Modes	user mode, kernel mode
Targets	live, crash dump
Platforms	all

Additional Information

On launch, the debugger will load all the settings from config.xml in the directory the debugger executable is in. Throughout your debugging session you can modify settings using the previous settings command (like .sympath or .prefer_dml) or the new .settings commands. You can use '.settings save' to save your settings to your settings configuration file. You can use the following command to enable AutoSave.

```
.settings set AutoSaveSettings=true
```

When auto save is enabled, the settings in the Debugger.Settings namespace will be automatically saved when exiting the debugger.

Remarks

You can exchange debug xml settings files with others to duplicate their debug settings.

.show_read_failures

12/5/2018 • 2 minutes to read • Edit Online

The **.show_read_failures** command enables or disables the display of read failures.

```
.show_read_failures /v
.show_sym_failures /V
```

Parameters

/v
Enables the display of read failures.

/V
Disables the display of read failures.

Environment

Modes	user mode, kernel mode
Targets	live, crash dump
Platforms	all

.show_sym_failures

12/5/2018 • 2 minutes to read • Edit Online

The **.show_sym_failures** command enables or disables the display of symbol lookup failures and type lookup failures.

```
.show_sym_failures /s
.show_sym_failures /S
.show_sym_failures /t
.show_sym_failures /T
```

Parameters

/s
Enables the display of symbol lookup failures.

/S
Disables the display of symbol lookup failures.

/t
Enables the display of type lookup failures.

/T
Disables the display of type lookup failures.

Environment

Modes	user mode, kernel mode
Targets	live, crash dump
Platforms	all

.sleep (Pause Debugger)

12/5/2018 • 2 minutes to read • Edit Online

The **.sleep** command causes the user-mode debugger to pause and the target computer to become active. This command is only used when you are controlling the user-mode debugger from the kernel debugger.

```
.sleep milliseconds
```

Parameters

milliseconds
Specifies the length of the pause, in milliseconds.

Environment

Modes	controlling the user-mode debugger from the kernel debugger
Targets	live debugging only
Platforms	all

Additional Information

For details and information about how to wake up a debugger in sleep mode, see Controlling the User-Mode Debugger from the Kernel Debugger.

Remarks

When you are controlling the user-mode debugger from the kernel debugger, and the user-mode debugger prompt is visible in the kernel debugger, this command will activate sleep mode. The kernel debugger, the user-mode debugger, and the target application will all freeze, but the rest of the target computer will become active.

If you use this command in any other scenario, it will simply freeze the debugger for a period of time.

The sleep time is in milliseconds and interpreted according to the default radix, unless a prefix such as **0n** is used. Thus, if the default radix is 16, the following command will cause about 65 seconds of sleep:

```
0:000> .sleep 10000
```

.sound_notify (Use Notification Sound)

12/5/2018 • 2 minutes to read • Edit Online

The **.sound_notify** command causes a sound to be played when WinDbg enters the wait-for-command state.

```
.sound_notify /ed
.sound_notify /ef File
.sound_notify /d
```

Parameters

/ed

Causes the default Windows alert sound to be played when WinDbg enters the wait-for-command state.

/ef **** *File*

Causes the sound contained in the specified file to be played when WinDbg enters the wait-for-command state.

/d

Disables the playing of sounds.

Environment

This command is available only in WinDbg and cannot be used in script files.

Modes	user mode, kernel mode
Targets	live, crash dump
Targets	all

.srcfix, .lsrcfix (Use Source Server)

12/5/2018 • 2 minutes to read • Edit Online

The **.srcfix** and **.lsrcfix** commands automatically set the source path to indicate that a source server will be used.

```
.srcfix[+] [Paths]
.lsrcfix[+] [Paths]
```

Parameters

+
Causes the existing source path to be preserved, and **; srv*** to be appended to the end. If the **+** is not used, the existing source path is replaced.

Paths
Specifies one or more additional paths to append to the end of the new source path.

Environment

The **.srcfix** command is available on all debuggers. The **.lsrcfix** command is available only in WinDbg and cannot be used in script files.

Modes	user mode, kernel mode
Targets	live, crash dump
Platforms	all

Additional Information

For more information on setting the local source path for a remote client, see **WinDbg Command-Line Options**. For details about SrcSrv, see Using a Source Server. For details on the source path and the local source path, see Source Path. For more information about commands that can be used while performing remote debugging through the debugger, see Controlling a Remote Debugging Session.

Remarks

When you add `srv*` to the source path, the debugger uses SrcSrv to retrieve source files from locations specified in the target modules' symbol files. Using `srv*` in the source path is fundamentally different from using `srv*` in the symbol path. In the symbol path, you can specify a symbol server location along with the `srv*` (for example, `.sympath SRV*https://msdl.microsoft.com/download/symbols`). In the source path, srv* stands alone, separated from all other elements by semicolons.

When this command is issued from a debugging client, **.srcfix** sets the source path to use a source server on the debugging server, while **.lsrcfix** does the same on the local machine.

These commands are the same as the **.srcpath (Set Source Path)** and **.lsrcpath (Set Local Source Path)** commands followed by the **srv*** source path element. Thus, the following two commands are equivalent:

```
.srcfix[+] [Paths]
.srcpath[+] srv*[;Paths]
```

Similarly, the following two commands are equivalent:

```
.lsrcfix[+] [Paths]
.lsrcpath[+] srv*[;Paths]
```

.srcnoisy (Noisy Source Loading)

12/5/2018 • 2 minutes to read • Edit Online

The **.srcnoisy** command controls the verbosity level for source file loading.

```
.srcnoisy [Options]
```

Parameters

Options
Can be any one of the following options:

0
Disables the display of extra messages.

1
Displays information about the progress of source file loading and unloading.

2
Displays information about the progress of symbol file loading and unloading.

3
Displays all information displayed by options 1 and 2.

Environment

Modes	user mode, kernel mode
Targets	live, crash dump
Platforms	all

Remarks

With no parameters, **.srcnoisy** will display the current status of noisy source loading.

Noisy source loading should not be confused with noisy symbol loading -- that is controlled by the **!sym noisy** extension and by other means of controlling the SYMOPT_DEBUG setting.

.srcpath, .lsrcpath (Set Source Path)

12/5/2018 • 2 minutes to read • Edit Online

The **.srcpath** and **.lsrcpath** commands set or display the source file search path.

```
.srcpath[+] [Directory [; ...]]
.lsrcpath[+] [Directory [; ...]]
```

Parameters

+
Specifies that the new directories will be appended to (rather than replacing) the previous source file search path.

Directory
Specifies one or more directories to put in the search path. If *Directory* is not specified, the current path is displayed. Separate multiple directories with semicolons.

Environment

The **.srcpath** command is available on all debuggers. The **.lsrcpath** command is available only in WinDbg and cannot be used in script files.

Modes	user mode, kernel mode
Targets	live, crash dump
Platforms	all

Additional Information

For details and other ways to change this path, see Source Path. For more information about commands that can be used while performing remote debugging through the debugger, see Controlling a Remote Debugging Session.

Remarks

If you include `srv*` in your source path, the debugger uses SrcSrv to retrieve source files from locations specified in the target modules' symbol files. For more information about using srv* in a source path, see Using a Source Server and **.srcfix**.

When this command is issued from a debugging client, **.srcpath** sets the source path on the debugging server, while **.lsrcpath** sets the source path on the local machine.

.step_filter (Set Step Filter)

12/5/2018 • 2 minutes to read • Edit Online

The **.step_filter** command creates a list of functions that are skipped (stepped over) when tracing. This allows you to trace through code and skip only certain functions. It can also be used in source mode to control stepping when there are multiple function calls on one line.

```
.step_filter "FilterList"
.step_filter /c
.step_filter
```

Parameters

"*FilterList***"**
Specifies the symbols associated with functions to be stepped over. *FilterList* can contain any number of text patterns separated by semicolons. Each of these patterns may contain a variety of wildcards and specifiers; see String Wildcard Syntax for details. A function whose symbol matches at least one of these patterns will be stepped over during tracing. Each time **"***FilterList***"** is used, any previous filter list is discarded and completely replaced with the new list.

/c
Clears the filter list.

Environment

Modes	user mode, kernel mode
Targets	live, crash dump
Platforms	all

Remarks

Without any parameters, **.step_filter** displays the current filter list.

Typically, a trace command (for example, **t** or the windbg debug | step into button) traces into a function call. However, if the symbol associated with the function being called matches a pattern specified by *FilterList*, the function will be stepped over -- as if a step command (for example, **p**) had been used.

If the instruction pointer is located within code that is listed in the filter list, any trace or step commands will step out of this function, like the **gu** command or the WinDbg **Step Out** button. Of course, this filter would prevent such code from having been traced into in the first place, so this will only happen if you have changed the filter or hit a breakpoint.

For example, the following command will cause trace commands to skip over all CRT calls:

```
.step_filter "msvcrt!*"
```

The **.step_filter** command is most useful when you are debugging in source mode, because there can be multiple function calls on a single source line. The **p** and **t** commands cannot be used to separate these function calls.

For example, in the following line, the **t** command will step into both GetTickCount and printf, while the **p** command will step over both function calls:

```
printf( "%x\n", GetTickCount() );
```

The **.step_filter** command allows you to filter out one of these calls while still tracing into the other.

Because the functions are identified by symbol, a single filter can include an entire module. This lets you filter out framework functions -- for example, Microsoft Foundation Classes (MFC) or Active Template Library (ATL) calls.

When debugging in assembly mode, each call is on a different line, so you can choose whether to step or trace line-by-line. So **.step_filter** is not very useful in assembly mode.

.suspend_ui (Suspend WinDbg Interface)

12/5/2018 • 2 minutes to read • Edit Online

The **.suspend_ui** command suspends the refresh of WinDbg debugging information windows.

```
.suspend_ui 0
.suspend_ui 1
.suspend_ui
```

Parameters

0

Suspends the refresh of WinDbg debugging information windows.

1

Enables the refresh of WinDbg debugging information windows.

Environment

This command is available only in WinDbg and cannot be used in script files.

Modes	kernel mode only
Targets	live, crash dump
Platforms	all

Additional Information

For information about debugging information windows, see Using Debugging Information Windows.

Remarks

Without any parameters, **.suspend_ui** displays whether debugging information windows are currently suspended.

By default, debugging information windows are refreshed every time the information they display changes.

Suspending the refresh of these windows can speed up WinDbg during a sequence of quick operations -- for example, when tracing or stepping many times in quick succession.

.symfix (Set Symbol Store Path)

12/5/2018 • 2 minutes to read • Edit Online

The **.symfix** command automatically sets the symbol path to point to the Microsoft symbol store.

```
.symfix[+] [LocalSymbolCache]
```

Parameters

+
Causes the Microsoft symbol store path to be appended to the existing symbol path. If this is not included, the existing symbol path is replaced.

LocalSymbolCache
Specifies the directory to be used as a local symbol cache. If this directory does not exist, it will be created when the symbol server begins copying files. If *LocalSymbolCache* is omitted, the sym subdirectory of the debugger installation directory will be used.

Environment

Modes	user mode, kernel mode
Targets	live, crash dump
Platforms	all

Additional Information

For details, see Using Symbol Servers and Symbol Stores.

Remarks

The following example shows how to use **.symfix** to set a new symbol path that points to the Microsoft symbol store.

```
3: kd> .symfix c:\myCache
3: kd> .sympath
Symbol search path is: srv*
Expanded Symbol search path is: cache*c:\myCache;SRV*https://msdl.microsoft.com/download/symbols
```

The following example shows how to use **.symfix+** to append the existing symbol path with a path that points to the Microsoft symbol store.

```
3: kd> .sympath
Symbol search path is: c:\someSymbols
Expanded Symbol search path is: c:\somesymbols
3: kd> .symfix+ c:\myCache
3: kd> .sympath
Symbol search path is: c:\someSymbols;srv*
Expanded Symbol search path is:
c:\somesymbols;cache*c:\myCache;SRV*https://msdl.microsoft.com/download/symbols
```

.symopt (Set Symbol Options)

12/5/2018 • 2 minutes to read • Edit Online

The **.symopt** command sets or displays the symbol options.

```
.symopt+ Flags
.symopt- Flags
.symopt
```

Parameters

+
Causes the symbol options specified by *Flags* to be set. If **.symopt** is used with *Flags* but no plus or minus sign, a plus sign is assumed.

-
Causes the symbol options specified by *Flags* to be cleared.

Flags
Specifies the symbol options to be changed. *Flags* must be the sum of the bit flags of these symbol options.

Environment

Modes	user mode, kernel mode
Targets	live, crash dump
Platforms	all

Additional Information

For a list and description of each symbol option, its bit flag, and other methods of setting and clearing these options, see Setting Symbol Options.

Remarks

Without any arguments, **.symopt** displays the current symbol options.

.sympath (Set Symbol Path)

12/5/2018 • 2 minutes to read • Edit Online

The **.sympath** command sets or alters the symbol path. The symbol path specifies locations where the debugger looks for symbol files.

```
.sympath[+] [Path [; ...]]
```

Parameters

+
Specifies that the new locations will be appended to (rather than replace) the previous symbol search path.

Path
A fully qualified path or a list of fully qualified paths. Multiple paths are separated by semicolons. If *Path* is omitted, the current symbol path is displayed.

Environment

Modes	user mode, kernel mode
Targets	live, crash dump
Platforms	all

Additional Information

For details and other ways to change this path, see Symbol Path.

Remarks

New symbol information will not be loaded when the symbol path is changed. You can use the **.reload (Reload Module)** command to reload symbols.

.thread (Set Register Context)

12/5/2018 • 4 minutes to read • Edit Online

The **.thread** command specifies which thread will be used for the register context.

```
.thread [/p [/r] ] [/P] [/w] [Thread]
```

Parameters

/p
(Live debugging only) If this option is included and *Thread* is nonzero, all transition page table entries (PTEs) for the process owning this thread will be automatically translated into physical addresses before access. This may cause slowdowns, because the debugger will have to look up the physical addresses for all the memory used by this process, and a significant amount of data may need to be transferred across the debug cable. (This behavior is the same as that of **.cache forcedecodeuser**.)

If the **/p** option is included and *Thread* is zero or omitted, this translation will be disabled. (This behavior is the same as that of **.cache noforcedecodeuser**.)

/r
(Live debugging only) If the **/r** option is included along with the **/p** option, user-mode symbols for the process owning this thread will be reloaded after the process and register contexts have been set. (This behavior is the same as that of **.reload /user**.)

/P
(Live debugging only) If this option is included and *Thread* is nonzero, all transition page table entries (PTEs) will be automatically translated into physical addresses before access. Unlike the **/p** option, this translates the PTEs for all user-mode and kernel-mode processes, not only the process owning this thread. This may cause slowdowns, because the debugger will have to look up the physical addresses for all memory in use, and a huge amount of data may need to be transferred across the debug cable. (This behavior is the same as that of **.cache forcedecodeptes**.)

/w
(64-bit kernel debugging only) Changes the active context for the thread to the WOW64 32-bit context. The thread specified must be running in a process that has a WOW64 state.

Thread
The address of the thread. If this is omitted or zero, the thread context is reset to the current thread.

Environment

Modes	kernel mode only
Targets	live, crash dump
Platforms	all

Additional Information

For more information about the register context and other context settings, see Changing Contexts.

Remarks

Generally, when you are doing kernel debugging, the only visible registers are the ones associated with the current thread.

The **.thread** command instructs the kernel debugger to use the specified thread as the register context. After this command is executed, the debugger will have access to the most important registers and the stack trace for this thread. This register context persists until you allow the target to execute or use another register context command (**.thread**, **.cxr**, or **.trap**). See Register Context for full details.

The **/w** option can only be used in 64-bit kernel debugging sessions on a thread running in a process that has a WOW64 state. The context retrieved will be the last context remembered by WOW64; this is usually the last user-mode code executed by *Thread*. This option can only be used if the target is in native machine mode. For example, if the target is running on a 64-bit machine that is emulating an x86-based processor using WOW64, this option cannot be used. Using the **/w** option will cause the machine mode to switch automatically to an x86-based processor.

This command does not actually change the current thread. In other words, extensions such as **!thread** and **!teb** will still default to the current thread if no arguments are used with them.

Here is an example. Use the **!process** extension to find the address of the desired thread. (In this case, **!process 0 0** is used to list all processes, then **!process** is used a second time to list all the threads for the desired process.)

```
kd> !process 0 0
**** NT ACTIVE PROCESS DUMP ****
PROCESS fe5039e0  SessionId: 0  Cid: 0008    Peb: 00000000  ParentCid: 0000
    DirBase: 00030000  ObjectTable: fe529a88  TableSize: 145.
    Image: System

.....

PROCESS ffaa5280  SessionId: 0  Cid: 0120    Peb: 7ffdf000  ParentCid: 01e0
    DirBase: 03b70000  ObjectTable: ffaa4e48  TableSize:  23.
    Image: winmine.exe

kd> !process ffaa5280
PROCESS ffaa5280  SessionId: 0  Cid: 0120    Peb: 7ffdf000  ParentCid: 01e0
    DirBase: 03b70000  ObjectTable: ffaa4e48  TableSize:  23.
    Image: winmine.exe
    VadRoot ffaf6e48 Clone 0 Private 50. Modified 0. Locked 0.
    DeviceMap fe502e88
    Token                             e1b55d70

.....

        THREAD ffaa43a0  Cid 120.3a4  Teb: 7ffde000  Win32Thread: e1b4fea8 WAIT: (WrUserRequest) UserMode
Non-Alertable
            ffadc6a0  SynchronizationEvent
        Not impersonating
        Owning Process ffaa5280
        WaitTime (seconds)      24323
        Context Switch Count    494                  LargeStack

.....
```

Now use the **.thread** command with the address of the desired thread. This sets the register context and enables you to examine the important registers and the call stack for this thread.

```
kd> .thread ffaa43a0
Using context of thread ffaa43a0

kd> r
Last set context:
eax=00000000 ebx=00000000 ecx=00000000 edx=00000000 esi=00000000 edi=00000000
eip=80403a0d esp=fd581c2c ebp=fd581c60 iopl=0         nv up di pl nz na pe nc
cs=0000  ss=0000  ds=0000  es=0000  fs=0000  gs=0000             efl=00000000
0000:3a0d ??              ???

kd> k
  *** Stack trace for last set context - .thread resets it
ChildEBP RetAddr
fd581c38 8042d61c ntoskrnl!KiSwapThread+0xc5
00001c60 00000000 ntoskrnl!KeWaitForSingleObject+0x1a1
```

.time (Display System Time)

12/5/2018 • 2 minutes to read • Edit Online

The **.time** command displays information about the system time variables.

```
.time [-h Hours]
```

Parameters

-h **** *Hours*
Specifies the offset from Greenwich Mean Time, in hours. A negative value of *Hours* must be enclosed in parentheses.

Environment

Modes	user mode, kernel mode
Targets	live, crash dump
Platforms	all

Remarks

The system time variables control performance counter behavior.

Here is an example in kernel mode:

```
kd> .time
Debug session time: Wed Jan 31 14:47:08 2001
System Uptime: 0 days 2:53:56
```

Here is an example in user mode:

```
0:000> .time
Debug session time: Mon Apr 07 19:10:50 2003
System Uptime: 4 days 4:53:56.461
Process Uptime: 0 days 0:00:08.750
  Kernel time: 0 days 0:00:00.015
  User time: 0 days 0:00:00.015
```

.tlist (List Process IDs)

12/5/2018 • 2 minutes to read • Edit Online

The **.tlist** command lists all processes running on the system.

```
.tlist [Options][FileNamePattern]
```

Parameters

Options
Can be any number of the following options:

-v
Causes the display to be verbose. This includes the session number, the process user name, and the command-line used to start the process.

-c
Limits the display to just the current process.

FileNamePattern
Specifies the file name to be displayed, or a pattern that the file name of the process must match. Only those processes whose file names match this pattern will be displayed. *FileNamePattern* may contain a variety of wildcards and specifiers; see String Wildcard Syntax for details. This match is made only against the actual file name, not the path.

Environment

Modes	user mode only
Targets	live debugging only
Platforms	all

Additional Information

For other methods of displaying processes, see Finding the Process ID.

Remarks

Each process ID is displayed with an **0n** prefix, to emphasize that the PID is a decimal number.

.trap (Display Trap Frame)

12/5/2018 • 2 minutes to read • Edit Online

The **.trap** command displays the trap frame register state and also sets the register context.

```
.trap [Address]
```

Parameters

Address
Hexadecimal address of the trap frame on the target system. Omitting the address does not display any trap frame information, but it does reset the register context.

Environment

Modes	kernel mode only
Targets	live, crash dump
Platforms	all

Additional Information

For more information about the register context and other context settings, see Changing Contexts.

Remarks

The **.trap** command displays the important registers for the specified trap frame.

This command also instructs the kernel debugger to use the specified context record as the register context. After this command is executed, the debugger will have access to the most important registers and the stack trace for this thread. This register context persists until you allow the target to execute or use another register context command (**.thread**, **.cxr**, or **.trap**). See Register Context for full details.

This extension is commonly used when debugging bug check 0xA and 0x7F. For details and an example, see **Bug Check 0xA** (IRQL_NOT_LESS_OR_EQUAL).

.tss (Display Task State Segment)

12/5/2018 • 2 minutes to read • Edit Online

The **.tss** command displays a formatted view of the saved Task State Segment (TSS) information for the current processor.

```
.tss [Address]
```

Parameters

Address
Address of the TSS.

Environment

Modes	kernel mode only
Targets	live, crash dump
Platforms	x86 only

Remarks

The address of the TSS can be found by examining the output of the **!pcr** extension.

.ttime (Display Thread Times)

12/5/2018 • 2 minutes to read • Edit Online

The **.ttime** command displays the running times for a thread.

```
.ttime
```

Environment

Modes	user mode only
Targets	live, crash dump
Platforms	x86 only

Remarks

This command only works in user mode. In kernel mode you should use **!thread** instead. This command works with user-mode minidumps as long as they were created with the **/mt** or **/ma** options; see User-Mode Dump Files for details.

The **.ttime** command shows the creation time of the thread, as well as the amount of time it has been running in kernel mode and in user mode.

Here is an example:

```
0:000> .ttime
Created: Sat Jun 28 17:58:42 2003
Kernel:  0 days 0:00:00.131
User:    0 days 0:00:02.109
```

.typeopt (Set Type Options)

6/20/2019 • 2 minutes to read • Edit Online

The **.typeopt** command sets or displays the type options.

```
.typeopt +Flags
.typeopt -Flags
.typeopt +FlagName
.typeopt -FlagName
.typeopt
```

Parameters

+
Causes the type option(s) specified by *Flags* or *FlagName* to be set.

-
Causes the type option(s) specified by *Flags* or *FlagName* to be cleared.

Flags
Specifies the type options to be changed. *Flags* can be a sum of any of the following values (there is no default):

0x1
Displays unsigned 16-bit integer values in all Watch windows and the Locals window as having UNICODE data type.

0x2
Displays signed 32-bit integer values in all Watch windows and the Locals window as unsigned integers in the default radix.

0x4
Displays signed integers of all sizes in all Watch windows and the Locals window as unsigned values in the default radix.

0x8
Causes the debugger to choose the matching symbol with the largest size when the Locals window or Watch window references a symbol by name but there is more than one symbol that matches this name. The size of a symbol is defined as follows: if the symbol is the name of a function, its size is the size of the function in memory. Otherwise, the size of the symbol is the size of the data type that it represents.

FlagName
Specifies the type options to be changed. *FlagName* can be any one of the following strings (there is no default):

uni
Displays unsigned 16-bit integer values in all Watch windows and the Locals window as having UNICODE data type. (This has the same effect as **0x1**.)

longst
Displays signed 32-bit integer values in all Watch windows and the Locals window as unsigned integers in the default radix. (This has the same effect as **0x2**.)

radix
Displays signed integers of all sizes in all Watch windows and the Locals window as unsigned values in the default

radix. (This has the same effect as **0x4**.)

size
Causes the debugger to choose the matching symbol with the largest size when the Locals window or Watch window references a symbol by name but there is more than one symbol that matches this name. The size of a symbol is defined as follows: if the symbol is the name of a function, its size is the size of the function in memory. Otherwise, the size of the symbol is the size of the data type that it represents. (This has the same effect as **0x8**.)

Environment

Modes	user mode, kernel mode
Targets	live, crash dump
Platforms	all

Remarks

Without any arguments, **.typeopt** displays the current symbol options.

To change the default radix, use the **n (Set Number Base)** command.

.unload (Unload Extension DLL)

12/5/2018 • 2 minutes to read • Edit Online

The **.unload** command unloads an extension DLL from the debugger.

```
.unload DLLName
!DLLName.unload
```

Parameters

DLLName
Specifies the file name of the debugger extension DLL to be unloaded. If the full path was specified when the DLL was loaded, it needs to be given in full here as well. If *DLLName* is omitted, the current extension DLL is unloaded.

Environment

Modes	user mode, kernel mode
Targets	live, crash dump
Platforms	all

Additional Information

For more details on loading, unloading, and controlling extensions, see Loading Debugger Extension DLLs.

Remarks

This command is useful when testing an extension you are creating. When the extension is recompiled, you must unload and then load the new DLL.

.unloadall (Unload All Extension DLLs)

12/5/2018 • 2 minutes to read • Edit Online

The **.unloadall** command unloads all extension DLLs from the debugger on the host system.

```
.unloadall
```

Environment

Modes	user mode, kernel mode
Targets	live, crash dump
Platforms	all

Additional Information

For more details on loading, unloading, and controlling extensions, see Loading Debugger Extension DLLs.

.urestart (Unregister for Restart)

12/5/2018 • 2 minutes to read • Edit Online

The **.urestart** command unregisters the debugging session for restart in case of a reboot or an application failure.

```
.urestart
```

Remarks

This command does not work for elevated debugger sessions.

See also

.rrestart

.wake (Wake Debugger)

12/5/2018 • 2 minutes to read • Edit Online

The **.wake** command causes sleep mode to end. This command is used only when you are controlling the user-mode debugger from the kernel debugger.

```
.wake PID
```

Parameters

PID
The process ID of the user-mode debugger.

Environment

Modes	controlling the user-mode debugger from the kernel debugger
Targets	live debugging only
Platforms	all

Additional Information

For more details, see Controlling the User-Mode Debugger from the Kernel Debugger. For information about how to find the process ID of the debugger, see Finding the Process ID.

Remarks

When you are controlling the user-mode debugger from the kernel debugger and the system is in sleep mode, this command can be used to wake up the debugger before the sleep timer runs out.

This command is not issued in the user-mode debugger on the target machine, nor in the kernel debugger on the host machine. It must be issued from a third debugger (KD, CDB, or NTSD) running on the target machine.

This debugger can be started expressly for this purpose, or can be another debugger that happens to be running. However, if there is no other debugger already running, it is easier just to use CDB with the **-wake command-line option**.

.write_cmd_hist (Write Command History)

12/5/2018 • 2 minutes to read • Edit Online

The **.write_cmd_hist** command writes the entire history of the Debugger Command window to a file.

```
.write_cmd_hist Filename
```

Parameters

Filename
Specifies the path and filename of the file that will be created.

Environment

This command is available only in WinDbg and cannot be used in script files.

Modes	user mode, kernel mode
Targets	live, crash dump
Platforms	all

.writemem (Write Memory to File)

12/5/2018 • 2 minutes to read • Edit Online

The **.writemem** command writes a section of memory to a file.

```
.writemem FileName Range
```

Parameters

FileName
Specifies the name of the file to be created. You can specify a full path and file name, or just the file name. If the file name contains spaces, *FileName* should be enclosed in quotation marks. If no path is specified, the current directory is used.

Range
Specifies the memory range to be written to the file. For syntax details, see Address and Address Range Syntax.

Environment

Modes	user mode, kernel mode
Targets	live, crash dump
Platforms	all

Remarks

The memory is copied literally to the file. It is not parsed in any way.

The **.writemem** command is the opposite of the **.readmem (Read Memory from File)** command.

.wtitle (Set Window Title)

12/5/2018 • 2 minutes to read • Edit Online

The **.wtitle** command sets the title in the main WinDbg window or in the NTSD, CDB, or KD window.

```
.wtitle Title
```

Parameters

Title
The title to use for the window.

Environment

This command cannot be used in script files.

Modes	user mode, kernel mode
Targets	live, crash dump
Platforms	all

Remarks

For CDB, NTSD, or KD, if the **.wtitle** command has not been used, the window title matches the command line used to launch the debugger.

For WinDbg, if **.wtitle** has not been used, the main window title includes the name of the target. If a debugging server is active, its connection string is displayed as well. If multiple debugging servers are active, only the most recent one is displayed.

When **.wtitle** is used, *Title* replaces all this information. Even if a debugging server is started later, *Title* will not change.

The WinDbg version number is always displayed in the window title bar, regardless of whether this command is used.

Control Keys

12/5/2018 • 2 minutes to read • Edit Online

This section of the reference discusses the various control keys that can be used in the debuggers.

These control keys work in KD and, in some cases, CDB. Many of these also work in WinDbg (using the CTRL+ALT keys instead of just CTRL).

WinDbg also uses the CTRL key, the ALT key, and the F keys as shortcut keys to toggle the menu options. See Shortcut Keys for a list of their meanings.

CTRL+\ (Debug Current Debugger)

12/5/2018 • 2 minutes to read • Edit Online

The **CTRL+** key launches a new instance of CDB; this new debugger takes the current debugger as its target.

```
CTRL+\  ENTER
```

Environment

Debuggers	CDB, NTSD, KD
Modes	user mode, kernel mode
Targets	live, crash dump
Platforms	all

Remarks

This is equivalent to launching a new CDB through the **remote.exe** utility, and using it to debug the debugger that you are already running.

CTRL+ is similar to the **.dbgdbg (Debug Current Debugger)** command.

CTRL+ALT+\ (Debug Current Debugger)

12/5/2018 • 2 minutes to read • Edit Online

The **CTRL+ALT+** key launches a new instance of CDB; this new debugger takes the current debugger as its target.

```
CTRL+ALT+\
```

Environment

Debuggers	WinDbg
Modes	user mode, kernel mode
Targets	live, crash dump
Platforms	all

Remarks

This is equivalent to launching a new CDB through the **remote.exe** utility, and using it to debug the debugger that you are already running.

CTRL+ALT+ is similar to the **.dbgdbg (Debug Current Debugger)** command, however **CTRL+ALT+** has the advantage that it can be used when no debugger command line is available.

CTRL+A (Toggle Baud Rate)

12/5/2018 • 2 minutes to read • Edit Online

The CTRL+A key toggles the baud rate used in the kernel debugging connection.

KD Syntax

```
CTRL+A  ENTER
```

WinDbg Syntax

```
CTRL+ALT+A
```

Environment

Debuggers	KD and WinDbg only
Modes	kernel mode only
Targets	live debugging only
Platforms	all

Remarks

This will cycle through all available baud rates for the kernel debugging connection.

Supported baud rates are 19200, 38400, 57600, and 115200. Each time this control key is used, the next baud rate will be selected. If the baud rate is already at 115200, it will be reduced to 19200.

In WinDbg, this can also be accomplished by selecting Debug | Kernel Connection | Cycle Baud Rate.

You cannot actually use this control key to change the baud rate at which you are debugging. The baud rate of the host computer and the target computer must match, and the baud rate of the target computer cannot be changed without rebooting. Therefore, you only need to toggle through the baud rates if the two computers are attempting to communicate at different rates. In this case, you must change the host computer's baud rate to match that of the target computer.

CTRL+B (Quit Local Debugger)

12/5/2018 • 2 minutes to read • Edit Online

The CTRL+B key causes the debugger to terminate abruptly. This does not end a remote debugging session.

```
CTRL+B  ENTER
```

Environment

Debuggers	CDB and KD only
Modes	user mode, kernel mode
Targets	live, crash dump
Platforms	all

Remarks

In CDB, the **q (Quit)** command should be used to exit. CTRL+B should only be used if the debugger is not responding.

In KD, the **q** command will end the debugging session and leave the target computer locked. If you need to preserve the debugging session (so a new debugger can connect to it), or if you need to leave the target computer running, you should use CTRL+B.

In WinDbg, the equivalent command is File | Exit or ALT+F4.

CTRL+C (Break)

12/5/2018 • 2 minutes to read • Edit Online

The CTRL+C key breaks into the debugger, stopping the target application or target computer, and cancels debugger commands.

CDB Syntax

```
CTRL+C
```

KD Syntax

```
CTRL+C
```

Target Computer Syntax

```
SYSRQ
ALT+SYSRQ
F12
```

Environment

Debuggers	CDB and KD only
Modes	user mode, kernel mode
Targets	live, crash dump
Platforms	all

Additional Information

For other methods of issuing this command and an overview of related commands, see Controlling the Target.

Remarks

When Using CDB:

In user mode, the CTRL+C key causes the target application to break into the debugger. The target application freezes, the debugger becomes active, and debugger commands can be entered.

If the debugger is already active, CTRL+C does not affect the target application. It can be, however, used to terminate a debugger command. For instance, if you have requested a long display and no longer want to see it, CTRL+C will end the display and return you to the debugger command prompt.

When performing remote debugging with CDB, CTRL+C can be pressed on the host computer's keyboard. If you want to issue a break from the target computer's keyboard, use CTRL+C on an x86 machine.

The F12 key can be used to get a command prompt when the application being debugged is busy. Set the focus on one of the target application's windows and press the F12 key on the target computer.

When Using KD:

In kernel mode, the CTRL+C key causes the target computer to break into the debugger. This locks the target computer and wakes up the debugger.

When debugging a system that is still running, CTRL+C must be pressed on the host keyboard to get the initial command prompt.

If the debugger is already active, CTRL+C does not affect the target computer. It can be used, however, to terminate a debugger command. For instance, if you have requested a long display and no longer want to see it, CTRL+C will end the display and return you to the debugger command prompt.

CTRL+C can also be used to get a command prompt when a debugger command is generating a long display or when the target computer is busy. When debugging an x86 machine, it can be pressed on either the host or target keyboard.

SYSRQ (or ALT+SYSRQ on an enhanced keyboard) is similar. It works from the host or target keyboard on any processor. However, it only works if the prompt has been acquired by pressing CTRL+C at least once before.

The SYSRQ key can be disabled by editing the registry. In the registry key

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\i8042prt\Parameters
```

create a value named **BreakOnSysRq**, and set it equal to DWORD 0x0. Then reboot. After this is done, pressing the SYSRQ key on the target computer's keyboard will not break into the kernel debugger.

When Debugging KD with CDB:

If you are debugging KD with CDB, then CTRL+C will be intercepted by the host debugger (CDB). To break into the target debugger (KD), you should use **CTRL+F** instead.

Note Note that in WinDbg, CTRL+C is a shortcut key that is used to copy text from a window. To issue a break command in WinDbg, use CTRL+BREAK or select Debug | Break from the menu.

CTRL+D (Toggle Debug Info)

12/5/2018 • 2 minutes to read • Edit Online

The CTRL+D key toggles debugger internal information flow on and off. This is used to restart communication between the target computer and the host computer in cases where the debugger is not communicating properly.

KD Syntax

```
CTRL+D  ENTER
```

WinDbg Syntax

```
CTRL+ALT+D
```

Environment

Debuggers	KD and WinDbg only
Modes	kernel mode only
Targets	live debugging only
Platforms	all

Remarks

When this is toggled on, the debugger outputs information about the communication between the host computer and the target computer.

This key can be used to test whether the debugger has crashed. If you suspect that either the debugger or the target is frozen, use this key. If you see packets being sent, the target is still working. If you see time-out messages, then the target is not responding. If there are no messages at all, the debugger has crashed.

If the target is not responding, use CTRL+R ENTER CTRL+C. If time-out messages continue to appear, the target has crashed (or the debugger was misconfigured).

This is also useful for debugging the KD debugger itself.

CTRL+F (Break to KD)

3/27/2019 • 2 minutes to read • Edit Online

The CTRL+F key cancels commands or breaks into the debugger. (This control key is particularly useful when you are using CDB to debug KD itself.)

```
CTRL+F   ENTER
```

Environment

Debuggers	CDB, KD
Modes	user mode, kernel mode
Targets	live, crash dump
Platforms	all

Remarks

Under typical conditions, CTRL+F is identical to the standard break commands (**CTRL+C** in KD and CDB, and Debug | Break or CTRL+BREAK in WinDbg.)

However, if you are debugging KD with CDB, these two keys will work differently. CTRL+C will be intercepted by the host debugger (CDB), while CTRL+F will be intercepted by the target debugger (KD).

See also

.breakin (Break to the Kernel Debugger)

CTRL+K (Change Post-Reboot Break State)

12/5/2018 • 2 minutes to read • Edit Online

The CTRL+K key changes the conditions on which the debugger will automatically break into the target computer.

KD Syntax

```
CTRL+K   ENTER
```

WinDbg Syntax

```
CTRL+ALT+K
```

Environment

Debuggers	KD and WinDbg only
Modes	kernel mode only
Targets	live debugging only
Platforms	all

Additional Information

For an overview of related commands and an explanation of how the reboot process affects the debugger, see Crashing and Rebooting the Target Computer.

Remarks

This control key causes the kernel debugger to cycle through the following three states:

No break
In this state, the debugger will not break into the target computer unless you press **CTRL+C**.

Break on reboot
In this state, the debugger will break into a rebooted target computer after the kernel initializes. This is equivalent to starting KD or WinDbg with the **-b**command-line option.

Break on first module load
In this state, the debugger will break into a rebooted target computer after the first kernel module is loaded. (This will cause the break to occur earlier than in the **Break on reboot** option.) This is equivalent to starting KD or WinDbg with the **-d**command-line option.

When CTRL+K is used, the new break state is displayed.

In WinDbg, this can also be accomplished by selecting Debug | Kernel Connection | Cycle Initial Break.

CTRL+P (Debug Current Debugger)

12/5/2018 • 2 minutes to read • Edit Online

The **CTRL+P** command is obsolete and has been replaced by **CTRL+** and **CTRL+ALT+**.

CTRL+R (Re-synchronize)

12/5/2018 • 2 minutes to read • Edit Online

The CTRL+R key synchronizes with the target computer.

KD Syntax

```
CTRL+R  ENTER
```

WinDbg Syntax

```
CTRL+ALT+R
```

Environment

Debuggers	KD and WinDbg only
Modes	kernel mode only
Targets	live debugging only
Platforms	all

Additional Information

For other methods of re-establishing contact with the target, see Synchronizing with the Target Computer.

Remarks

This attempts to synchronize the host computer with the target computer. Use this key if the target is not responding.

If you are using a 1394 kernel connection, resynchronization may not always be successful.

CTRL+V (Toggle Verbose Mode)

12/5/2018 • 2 minutes to read • Edit Online

The CTRL+V key toggles verbose mode on and off.

CDB / KD Syntax

```
CTRL+V   ENTER
```

WinDbg Syntax

```
CTRL+ALT+V
```

Environment

Debuggers	CDB, KD, WinDbg
Modes	user mode, kernel mode
Targets	live, crash dump
Platforms	all

Remarks

When verbose mode is turned on, some display commands (such as register dumping) produce more detailed output. Every MODULE LOAD operation that is sent to the debugger will be displayed. And every time a driver or DLL is loaded by the operating system, the debugger will be notified.

In WinDbg, this can also be accomplished by selecting View | Verbose Output.

CTRL+W (Show Debugger Version)

4/3/2019 • 2 minutes to read • Edit Online

The CTRL+W key displays version information for the debugger and all loaded extension DLLs.

CDB / KD Syntax

```
CTRL+W  ENTER
```

WinDbg Syntax

```
CTRL+ALT+W
```

Environment

Debuggers	CDB, KD, WinDbg
Modes	user mode, kernel mode
Targets	live, crash dump
Platforms	all

Remarks

This has the same effect as the **version (Show Debugger Version)** command, except that the latter command displays the Windows operating system version as well.

In WinDbg, this can also be accomplished by selecting View | Show Version.

See also

version (Show Debugger Version)

vertarget (Show Target Computer Version)

General Extensions

12/5/2018 • 2 minutes to read • Edit Online

This section describes extension commands that are frequently used during both user-mode and kernel-mode debugging.

The debugger automatically loads the proper version of these extension commands. Unless you have manually loaded a different version, you do not have to keep track of the DLL versions that are being used. For more information about the default module search order, see Using Debugger Extension Commands. For more information about how to load extension modules, see Loading Debugger Extension DLLs.

Each extension command reference topics lists the DLLs that expose that command. Use the following rules to determine the proper directory to load this extension DLL from:

- If your target computer is running Microsoft Windows XP or a later version of Windows, use winxp\kdexts.dll, winxp\ntsdexts.dll, winxp\exts.dll, winext\ext.dll, or dbghelp.dll.

!acl

12/5/2018 • 2 minutes to read • Edit Online

The **!acl** extension formats and displays the contents of an access control list (ACL).

Syntax

```
!acl Address [Flags]
```

Parameters

Address
Specifies the hexadecimal address of the ACL.

Flags
Displays the friendly name of the ACL, if the value of *Flags* is 1. This friendly name includes the security identifier (SID) type and the domain and user name for the SID.

DLL

Exts.dll

Additional Information

For more information about access control lists, see **!sid**, **!sd**, and Determining the ACL of an Object. Also, see the Microsoft Windows SDK documentation, the Windows Driver Kit (WDK) documentation, and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

The following example shows the **!acl** extension.

```
kd> !acl e1bf35d4 1
ACL is:
ACL is: ->AclRevision: 0x2
ACL is: ->Sbz1       : 0x0
ACL is: ->AclSize    : 0x40
ACL is: ->AceCount   : 0x2
ACL is: ->Sbz2       : 0x0
ACL is: ->Ace[0]: ->AceType: ACCESS_ALLOWED_ACE_TYPE
ACL is: ->Ace[0]: ->AceFlags: 0x0
ACL is: ->Ace[0]: ->AceSize: 0x24
ACL is: ->Ace[0]: ->Mask : 0x10000000
ACL is: ->Ace[0]: ->SID: S-1-5-21-518066528-515770016-299552555-2981724 (User: MYDOMAIN\myuser)

ACL is: ->Ace[1]: ->AceType: ACCESS_ALLOWED_ACE_TYPE
ACL is: ->Ace[1]: ->AceFlags: 0x0
ACL is: ->Ace[1]: ->AceSize: 0x14
ACL is: ->Ace[1]: ->Mask : 0x10000000
ACL is: ->Ace[1]: ->SID: S-1-5-18 (Well Known Group: NT AUTHORITY\SYSTEM)
```

!address

3/29/2019 • 5 minutes to read • Edit Online

The **!address** extension displays information about the memory that the target process or target computer uses.

User-Mode

```
!address Address
!address -summary
!address [-f:F1,F2,...] {[-o:{csv | tsv | 1}] | [-c:"Command"]}
!address -? | -help
```

Kernel-Mode

```
!address Address
!address
```

Parameters

Address
Displays only the region of the address space that contains *Address*.

-summary
Displays only summary information.

-f:*F1*, *F2*, ...
Displays only the regions specified by the filters *F1*, *F2*, and so on.

The following filter values specify memory regions by the way that the target process is using them.

FILTER VALUE	MEMORY REGIONS DISPLAYED
VAR	Busy regions. These regions include all virtual allocation blocks, the SBH heap, memory from custom allocators, and all other regions of the address space that fall into no other classification.
Free	Free memory. This includes all memory that has not been reserved.
Image	Memory that is mapped to a file that is part of an executable image.
Stack	Memory used for thread stacks.
Teb	Memory used for thread environment blocks (TEBs).

FILTER VALUE	MEMORY REGIONS DISPLAYED
Peb	Memory used for the process environment block (PEB).
Heap	Memory used for heaps.
PageHeap	The memory region used for the full-page heap.
CSR	CSR shared memory.
Actx	Memory used for activation context data.
NLS	Memory used for National Language Support (NLS) tables.
FileMap	Memory used for memory-mapped files. This filter is applicable only during live debugging.

The following filter values specify memory regions by the memory type.

FILTER VALUE	MEMORY REGIONS DISPLAYED
MEM_IMAGE	Memory that is mapped to a file that is part of an executable image.
MEM_MAPPED	Memory that is mapped to a file that is not part of an executable image. This includes memory that is mapped to the paging file.
MEM_PRIVATE	Private memory. This memory is not shared by any other process, and it is not mapped to any file.

The following filter values specify memory regions by the state of the memory.

FILTER VALUE	MEMORY REGIONS DISPLAYED
MEM_COMMIT	Committed memory.
MEM_FREE	Free memory. This includes all memory that has not been reserved.
MEM_RESERVE	Reserved memory.

The following filter values specify memory regions by the protection applied to the memory.

FILTER VALUE	MEMORY REGIONS DISPLAYED
PAGE_NOACCESS	Memory that cannot be accessed.
PAGE_READONLY	Memory that is readable, but not writable and not executable.
PAGE_READWRITE	Memory that is readable and writable, but not executable.
PAGE_WRITECOPY	Memory that has copy-on-write behavior.
PAGE_EXECUTE	Memory that is executable, but not readable and not writeable.
PAGE_EXECUTE_READ	Memory that is executable and readable, but not writable.
PAGE_EXECUTE_READWRITE	Memory that is executable, readable, and writable.
PAGE_EXECUTE_WRITECOPY	Memory that is executable and has copy-on-write behavior.
PAGE_GUARD	Memory that acts as a guard page.
PAGE_NOCACHE	Memory that is not cached.
PAGE_WRITECOMBINE	Memory that has write-combine access enabled.

-o:{**csv** | **tsv** | **1**}
Displays the output according to one of the following options.

OPTION	OUTPUT FORMAT
csv	Displays the output as comma-separated values.
tsv	Displays the output as tab-separated values.
1	Displays the output in bare format. This format works well when **!address** is used as input to **.foreach**.

-c:"*Command*"
Executes a custom command for each memory region. You can use the following placeholders in your command to represent output fields of the **!address** extension.

PLACEHOLDER	OUTPUT FIELD
%1	Base address
%2	End address + 1
%3	Region size
%4	Type
%5	State
%6	Protection
%7	Usage

For example, `!address -f:Heap -c:".echo %1 %3 %5"` displays the base address, size, and state for each memory region of type **Heap**.

Quotes in the command must be preceded by a back slash (\"). For example, !address -f:Heap -c:"s -a %1 %2 \"pad\"" searches each memory region of type **Heap** for the string "pad".

Multiple commands separated by semicolons are not supported.

-?
Displays minimal Help text for this extension in the Debugger Command window.

DLL

Windows 2000	Ext.dll
Windows XP and later	Ext.dll

Additional Information

For more information about how to display and search memory, see Reading and Writing Memory. For additional extensions that display memory properties, see **!vm** (kernel mode) and **!vprot** (user mode).

Remarks

Without any parameters, the **!address** extension displays information about the whole address space. The **!address -summary** command shows only the summary.

In kernel mode, this extension searches only kernel memory, even if you used **.process (Set Process Context)** to specify a given process' virtual address space. In user mode, the **!address** extension always refers to the memory that the target process owns.

In user mode, **!address** *Address* shows the characteristics of the region that the specified address belongs to. Without parameters, **!address** shows the characteristics of all memory regions. These characteristics include the memory usage, memory type, memory state, and memory protection. For more information about the meaning

of this information, see the earlier tables in the description of the **-f** parameter.

The following example uses **!address** to retrieve information about a region of memory that is mapped to kernel32.dll.

```
0:000> !address 75831234
Usage:                  Image
Base Address:           75831000
End Address:            758f6000
Region Size:            000c5000
Type:                   01000000MEM_IMAGE
State:                  00001000MEM_COMMIT
Protect:                00000020PAGE_EXECUTE_READ
More info:              lmv m kernel32
More info:              !lmi kernel32
More info:              ln 0x75831234
```

This example uses an *Address* value of 0x75831234. The display shows that this address is in a memory region that begins with the address 0x75831000 and ends with the address 0x758f6000. The region has usage **Image**, type **MEM_IMAGE**, state **MEM_COMMIT**, and protection **PAGE_EXECUTE_READ**. (For more information about the meaning of these values, see the earlier tables.) The display also lists three other debugger commands that you can use to get more information about this memory address.

If you are starting with an address and trying to determine information about it, the usage information is frequently the most valuable. After you know the usage, you can use additional extensions to learn more about this memory. For example, if the usage is **Heap**, you can use the **!heap** extension to learn more.

The following example uses the **s (Search Memory)** command to search each memory region of type **Image** for the wide-character string "Note".

```
!address /f:Image /c:"s -u %1 %2 \"Note\""

*** Executing: s -u 0xab0000 0xab1000 "Note"
*** Executing: s -u 0xab1000 0xabc000 "Note"
00ab2936  004e 006f 0074 0065 0070 0061 0064 0000  N.o.t.e.p.a.d...
00ab2f86  004e 006f 0074 0065 0070 0061 0064 005c  N.o.t.e.p.a.d.\.
00ab32e4  004e 006f 0074 0065 0070 0061 0064 0000  N.o.t.e.p.a.d...
*** Executing: s -u 0xabc000 0xabd000 "Note"
. . .
```

In kernel mode, the output of **!address** is similar to the user mode output but contains less information. The following example shows the kernel mode output.

```
kd> !address
  804de000 - 00235000
 Usage       KernelSpaceUsageImage
          ImageName   ntoskrnl.exe

  80c00000 - 001e1000
          Usage       KernelSpaceUsagePFNDatabase

....

  f85b0000 - 00004000
          Usage       KernelSpaceUsageKernelStack
          KernelStack 817b4da0 : 324.368

 f880d000 - 073d3000
          Usage       KernelSpaceUsageNonPagedPoolExpansion
```

The meaning of "usage" is the same as in user mode. "ImageName" indicates the module that is associated with this address. "KernelStack" shows the address of this thread's ETHREAD block (0x817B4DA0), the process ID (0x324), and the thread ID (0x368).

!analyze

12/5/2018 • 2 minutes to read • Edit Online

The **!analyze** extension displays information about the current exception or bug check.

User-Mode

```
!analyze [-v] [-f | -hang] [-D BucketID]
!analyze -c [-load KnownIssuesFile | -unload | -help ]
```

Kernel-Mode

```
!analyze [-v] [-f | -hang] [-D BucketID]
!analyze -c [-load KnownIssuesFile | -unload | -help ]
!analyze -show BugCheckCode [BugParameters]
```

Parameters

-v
Displays verbose output.

-f
Generates the **!analyze** exception output. Use this parameter to see an exception analysis even when the debugger does not detect an exception.

-hang
Generates **!analyze** hung-application output. Use this parameter when the target has experienced a bug check or exception, but an analysis of why an application hung is more relevant to your problem. In kernel mode, **!analyze -hang** investigates locks that the system holds and then scans the DPC queue chain. In user mode, **!analyze -hang** analyzes the thread stack to determine whether any threads are blocking other threads.

Before you run this extension in user mode, consider changing the current thread to the thread that you think has stopped responding (that is, hung), because the exception might have changed the current thread to a different one.

-D *BucketID*
Displays only those items that are relevant to the specified *BucketID*.

-show *BugCheckCode* [*BugParameters*]
Displays information about the bug check specified by *BugCheckCode*. *BugParameters* specifies up to four bug check parameters separated by spaces. These parameters enable you to further refine your search.

-c
Continues execution when the debugger encounters a known issue. If the issue is not a "known" issue, the debugger remains broken into the target.

You can use the **-c** option with the following subparameters. These subparameters configure the list of known issues. They do not cause execution to occur by themselves. Until you run **!analyze -c** **** **-load** at least one time, **!analyze -c** has no effect.

-load *KnownIssuesFile*
Loads the specified known-issues file. *KnownIssuesFile* specifies the path and file name of this file. This file must be in XML format. You can find a sample file in the sdk\samples\analyze_continue subdirectory of the debugger installation directory. (You must have performed a full installation of Debugging Tools for Windows to have this file.)

The list of known issues in the *KnownIssuesFile* file is used for all later **-c** commands until you use **-c** **** **-unload**, or until you use **-c** **** **-load** again (at which point the new data replaces the old data).

-unload
Unloads the current list of known issues.

-help
Displays help for the **!analyze -c** extension commands extension in the Debugger Command window.

DLL

Ext.dll

Additional Information

For sample analysis of a user-mode exception and of a kernel-mode stop error (that is, crash), and for more information about how **!analyze** uses the triage.ini file, see Using the !analyze Extension.

Remarks

In user mode, **!analyze** displays information about the current exception.

In kernel mode, **!analyze** displays information about the most recent bug check. If a bug check occurs, the **!analyze** display is automatically generated. You can use **!analyze -v** to show additional information. If you want to see only the basic bug check parameters, you can use the **.bugcheck (Display Bug Check Data)** command.

For drivers that use User-Mode Driver Framework (UMDF) version 2.15 or later, **!analyze** provides information about UMDF verifier failures and unhandled exceptions. This functionality is available when performing live kernel-mode debugging, as well when analyzing a user-mode memory dump file. For UMDF driver crashes, **!analyze** attempts to identify the responsible driver.

!asd

12/5/2018 • 2 minutes to read • Edit Online

The **!asd** extension displays a specified number of failure analysis entries from the data cache, starting at the specified address.

```
!asd Address DataUsed
```

Parameters

Address
Specifies the address of the first failure analysis entry to display.

DataUsed
Determines the number of tokens to display.

DLL

Windows 2000	Ext.dll
Windows XP and later	Ext.dll

Additional Information

You can use the **!dumpfa** extension to debug the **!analyze** extension.

Remarks

The **!asd** extension is useful only when you are debugging the **!analyze** extension.

!atom

1/3/2019 • 2 minutes to read • Edit Online

The **!atom** extension displays the formatted atom table for the specified atom or for all atoms of the current process.

```
!atom [Address]
```

Parameters

Address
Specifies the hexadecimal virtual address of the atom to display. If you omit this parameter or specify zero, the atom table for the current process is displayed. This table lists all atoms for the process.

DLL

Exts.dll

Additional Information

For more information about atoms and atom tables, see the Microsoft Windows SDK documentation.

!bitcount

12/5/2018 • 2 minutes to read • Edit Online

The **!bitcount** extension counts the number of "1" bits in a memory range.

```
!bitcount StartAddress TotalBits
```

Parameters

StartAddress
Specifies the starting address of the memory range whose "1" bits will be counted.

TotalBits
Specifies the size of the memory range, in bits.

-?
Displays some Help text for this extension in the Debugger Command window.

DLL

Windows 2000	Unavailable
Windows XP and later	Exts.dll

!blackboxbsd

7/13/2019 • 2 minutes to read • Edit Online

The **!blackboxbsd** extension displays cached Boot Status Data (BSD) information when it is available in a kernel mode dump file. The information is retrieved from cached data in the kernel mode dump which was saved at the time the bugcheck occurred, and may not always be available.

Syntax

```
!blackboxbsd
```

Parameters

None

DLL

ext.dll

Remarks

Driver developers can add secondary boot information to dump files. Driver developers (and the OS) can decide when to add this information to the dump file. This means that not all kernel mode dump files will contain secondary boot information. For more information, see Writing a Bug Check Reason Callback Routine.

Example Command Output

```
0: kd> !ext.blackboxbsd
Version: 136
Product type: 1

Auto advanced boot: FALSE
Advanced boot menu timeout: 30
Last boot succeeded: TRUE
Last boot shutdown: FALSE
Sleep in progrees: FALSE

Power button timestamp: 0
System running: TRUE
Connected standby in progress: FALSE
User shutdown in progress: FALSE
System shutdown in progress: FALSE
Sleep in progress: 6
Connected standby scenario instance id: 5
Connected standby entry reason: 12
Connected standby exit reason: 31
System sleep transitions to on: 8
Last reference time: 0x1d3645f716b79e3
Last reference time checksum: 0xb6ae84b7
Last update boot id: 6

Boot attempt count: 1
Last boot checkpoint: TRUE
Checksum: 0x7f
Last boot id: 6
Last successful shutdown boot id: 2
Last reported abnormal shutdown boot id: 5

Error info boot id: 0
Error info repeat count: 0
Error info other error count: 0
Error info code: 0
Error info other error count: 0

Power button last press time: 0x1d365b105eb9e3b
Power button cumulative press count: 6
Power button last press boot id: 6
Power button last power watchdog stage: 0x20
Power button watchdog armed: FALSE
Power button shutdown in progress: FALSE
Power button last release time: 0x1d36112993b1191
Power button cumulative release count: 5
Power button last release boot id: 6
Power button error count: 0
Power button current connected standby phase: 1
Power button transition latest checkpoint id: 9
Power button transition latest checkpoint type: 0
Power button transition latest checkpoint sequence number: 77
```

!blackboxscm

7/13/2019 • 2 minutes to read • Edit Online

The **!blackboxscm** extension displays information from Service Control Manager (SCM) when it is available in a kernel mode dump file. The extension will display the names of any services which have outstanding Service Control Requests. The information is retrieved from cached data in the kernel mode dump which was saved at the time the bugcheck occurred, and may not always be available.

Syntax

```
!blackboxscm
```

Parameters

None

DLL

ext.dll

Remarks

The commands have been dispatched to the LPHANDLER_FUNCTION_EX callback function of the specific service.
Outstanding requests such as SERVICE_CONTROL_SHUTDOWN or SERVICE_CONTROL_PRESHUTDOWN could be delaying the orderly shutdown or restart of the computer.

Example Command Output

In many dump files, just a single service is returned.

```
2: kd> !ext.blackboxscm
    Name: gpsvc
    Code: 15
```

The returned data provides information on two fields.

Name - The name of the service that was active at the time that the dump occurred.

Code - The Decimal value of the dwControl which is outstanding

In this example, the code 15 (or 0x0000000F) is defined as SERVICE_CONTROL_PRESHUTDOWN.

Multiple Services

When multiple services are listed, only the first service that is listed is typically of interest for failure analysis. This is because the SCM (Service Control Manager) waits serially for completion of these requests, so only the first service has actually received a control request.

For more information about SCM, see Service Control Manager.

Additional Information

dwControl values are defined in winsvc.h and documented as parameters to LPHANDLER_FUNCTION_EX

callback function.

!chksym

12/5/2018 • 2 minutes to read • Edit Online

The **!chksym** extension tests the validity of a module against a symbol file.

```
!chksym Module [Symbol]
```

Parameters

Module
Specifies the name of the module by its name or base address.

Symbol
Specifies the name of a symbol file.

DLL

Windows 2000	Unavailable
Windows XP	Unavailable
Windows Vista and later	Dbghelp.dll

Remarks

If you do not specify a symbol filed, the loaded symbol is tested. Otherwise, if you specify a .pdb or .dbg symbol file path, the loaded symbol is tested against the loaded module.

!chkimg

12/5/2018 • 9 minutes to read • Edit Online

The **!chkimg** extension detects corruption in the images of executable files by comparing them to the copy on a symbol store or other file repository.

```
!chkimg [Options] [-mmw LogFile LogOptions] [Module]
```

Parameters

Options
Any combination of the following options:

-p **** *SearchPath*
Recursively searches *SearchPath* for the file before accessing the symbol server.

-f
Fixes errors in the image. Whenever the scan detects differences between the file on the symbol store and the image in memory, the contents of the file on the symbol store are copied over the image. If you are performing live debugging, you can create a dump file before you execute the **!chkimg -f** extension.

-nar
Prevents the mapped image of the file on the symbol server from being moved. By default, when the copy of the file is located on the symbol server and mapped into memory, **!chkimg** moves the image of the file on the symbol server. However, if you use the **-nar** option, the image of the file from the server is not moved.

The executable image that is already in memory (that is, the one that is being scanned) is moved, because the debugger always relocates images that it loads.

This switch is useful only if the operating system already moved the original image. If the image has not been moved, **!chkimg** and the debugger will move the image. Use of this switch is rare.

-ss **** *SectionName*
Limits the scan to those sections whose names contain the string *SectionName*. The scan will include any non-discardable section whose name contains this string. *SectionName* is case sensitive and cannot exceed 8 characters.

-as
Causes the scan to include all sections of the image except discardable sections. By default, (if you do not use **-as** or **-ss**), the scan skips sections that are writeable, sections that are not executable, sections that have "PAGE" in their name, and discardable sections.

-r **** *StartAddress* **** *EndAddress*
Limits the scan to the memory range that begins with *StartAddress* and ends with *EndAddress*. Within this range, any sections that would typically be scanned are scanned. If a section partially overlaps with this range, only that part of the section that overlaps with this range is scanned. The scan is limited to this range even if you also use the **-as** or **-ss** switch.

-nospec
Causes the scan to include the reserved sections of Hal.dll and Ntoskrnl.exe. By default, **!chkimg** does not check certain parts of these files.

-noplock
Displays areas that mismatch by having a byte value of 0x90 (a **nop** instruction) and a byte value of 0xF0 (a **lock** instruction). By default, these mismatches are not displayed.

-np
Causes patched instructions to be recognized.

-d
Displays a summary of all mismatched areas while the scan is occurring. For more information about this summary text, see the Remarks section.

-db
Displays mismatched areas in a format that is similar to the **db debugger command**. Therefore, each display line shows the address of the first byte in the line, followed by up to 16 hexadecimal byte values. The byte values are immediately followed by the corresponding ASCII values. All nonprintable characters, such as carriage returns and line feeds, are displayed as periods (.). The mismatched bytes are marked by an asterisk (*).

-lo **** *lines*
Limits the number of output lines that **-d** or **-db** display to the lines number of lines.

-v
Displays verbose information.

-mmw
Creates a log file and records the activity of **!chkimg** in this file. Each line of the log file represents a single mismatch.

LogFile
Specifies the full path of the log file. If you specify a relative path, the path is relative to the current path.

LogOptions
Specifies the contents of the log file. *LogOptions* is a string that consists of a concatenation of various letters. Each line in the log file contains several columns that are separated by commas. These columns include the items that the following option letters specify, in the order that the letters appear in the *LogOptions* string. You can include the following options multiple times. You must include at least one option.

LOG OPTION	INFORMATION INCLUDED IN THE LOG FILE
v	The virtual address of the mismatch
r	The offset (relative address) of the mismatch within the module
s	The symbol that corresponds to the address of the mismatch
S	The name of the section that contains the mismatch
e	The correct value that was expected at the mismatch location
w	The incorrect value that was at the mismatch location

LogOptions can also include some, or none, of the following additional options.

LOG OPTION	EFFECT
o	If a file that has the name *LogFile* already exists, the existing file is overwritten. By default, the debugger appends new information to the end of any existing file.
t*String*	Adds an extra column to the log file. Each entry in this column contains *String*. The **t***String* option is useful if you are appending new information to an existing log file and you have to distinguish the new records from the old. You cannot add space between **t** and *String*. If you use the **t**|*String* option, it must be the final option in *LogOptions*, because *String* is taken to include all of the characters that are present before the next space.

For example, if *LogOptions* is **rSewo**, each line of the log file contains the relative address and section name of the mismatch location and the expected and actual values at that location. This option also causes any previous file to be overwritten. You can use the **-mmw** switch multiple times if you want to create several log files that have different options. You can create up to 10 log files at the same time.

Module
Specifies the module to check. *Module* can be the name of the module, the starting address of the module, or any address that is contained in the module. If you omit *Module*, the debugger uses the module that contains the current instruction pointer.

DLL

Windows 2000	Ext.dll
Windows XP and later	Ext.dll

Remarks

When you use **!chkimg**, it compares the image of an executable file in memory to the copy of the file that resides on a symbol store.

All sections of the file are compared, except for sections that are discardable, that are writeable, that are not executable, that have "PAGE" in their name, or that are from INITKDBG. You can change this behavior can by using the **-ss**, **-as**, or **-r** switches.

!chkimg displays any mismatch between the image and the file as an image error, with the following exceptions:

- Addresses that are occupied by the Import Address Table (IAT) are not checked.
- Certain specific addresses in Hal.dll and Ntoskrnl.exe are not checked, because certain changes occur when these sections are loaded. To check these addresses, include the **-nospec** option.
- If the byte value 0x90 is present in the file, and if the value 0xF0 is present in the corresponding byte of the image (or vice versa), this situation is considered a match. Typically, the symbol server holds one version of a binary that exists in both uniprocessor and multiprocessor versions. On an x86-based processor, the **lock** instruction is 0xF0, and this instruction corresponds to a **nop** (0x90) instruction in the uniprocessor version. If you want **!chkimg** to display this pair as a mismatch, set the **-noplock** option.

Note If you use the **-f** option to fix image mismatches, **!chkimg** fixes only those mismatches that it considers to be errors. For example, **!chkimg** does not change an 0x90 byte to an 0xF0 byte unless you include **-noplock**.

When you include the **-d** option, **!chkimg** displays a summary of all mismatched areas while the scan is occurring. Each mismatch is displayed on two lines. The first line includes the start of the range, the end of the range, the size of the range, the symbol name and offset that corresponds to the start of the range, and the number of bytes since the last error (in parentheses). The second line is enclosed in brackets and includes the hexadecimal byte values that were expected, a colon, and then the hexadecimal byte values that were actually encountered in the image. If the range is longer than 8 bytes, only the first 8 bytes are shown before the colon and after the colon. The following example shows this situation.

```
be000015-be000016  2 bytes - win32k!VeryUsefulFunction+15 (0x8)
     [ 85 dd:95 23 ]
```

Occasionally, a driver alters part of the Microsoft Windows kernel by using hooks, redirection, or other methods. Even a driver that is no longer on the stack might have altered part of the kernel. You can use the **!chkimg** extension as a file comparison tool to determine which parts of the Windows kernel (or any other image) are being altered by drivers and exactly how the parts are being changed. This comparison is most effective on full dump files.

You can also use **!chkimg** together with the **!for_each_module** extension to check the image of each loaded module. The following example shows this situation.

```
!for_each_module !chkimg @#ModuleName
```

Suppose that you encounter a bug check, for example, and begin by using **!analyze**.

```
kd> !analyze
....
BugCheck 1000008E, {c0000005, bf920e48, baf75b38, 0}
Probably caused by : memory_corruption
CHKIMG_EXTENSION: !chkimg !win32k
....
```

In this example, the **!analyze** output suggests that memory corruption has occurred and includes a CHKIMG_EXTENSION line that suggests that Win32k.sys could be the corrupted module. (Even if this line is not present, you might consider possible corruption in the module on top of the stack.) Start by using **!chkimg** without any switches, as the following example shows.

```
kd> !chkimg win32k
Number of different bytes for win32k: 31
```

The following example shows that there are indeed memory corruptions. Use **!chkimg -d** to display all of the errors for the Win32k module.

```
kd> !chkimg win32k -d
    bf920e40-bf920e46  7 bytes - win32k!HFDBASIS32::vSteadyState+1f
        [ 78 08 d3 78 0c c2 04:00 00 00 00 00 01 00 ]
    bf920e48-bf920e5f  24 bytes - win32k!HFDBASIS32::vHalveStepSize (+0x08)
        [ 8b 51 0c 8b 41 08 56 8b:00 00 00 00 00 00 00 00 ]
Number of different bytes for win32k: 31
```

When you try to disassemble the corrupted image of the second section that is listed, the following output might occur.

```
kd> u  win32k!HFDBASIS32::vHalveStepSize
win32k!HFDBASIS32::vHalveStepSize:
bf920e48 0000             add     [eax],al
bf920e4a 0000             add     [eax],al
bf920e4c 0000             add     [eax],al
bf920e4e 0000             add     [eax],al
bf920e50 7808            js win32k!HFDBASIS32::vHalveStepSize+0x12 (bf920e5a)
bf920e52 d3780c           sar     dword ptr [eax+0xc],cl
bf920e55 c20400           ret     0x4
bf920e58 8b510c           mov     edx,[ecx+0xc]
```

Then, use **!chkimg -f**to fix the memory corruption.

```
kd> !chkimg win32k -f
Warning: Any detected errors will be fixed to what we expect!
Number of different bytes for win32k: 31 (fixed)
```

Now you can disassemble the corrected view and see the changes that you have made.

```
kd> u  win32k!HFDBASIS32::vHalveStepSize
win32k!HFDBASIS32::vHalveStepSize:
bf920e48 8b510c           mov     edx,[ecx+0xc]
bf920e4b 8b4108           mov     eax,[ecx+0x8]
bf920e4e 56               push    esi
bf920e4f 8b7104           mov     esi,[ecx+0x4]
bf920e52 03c2             add     eax,edx
bf920e54 c1f803           sar     eax,0x3
bf920e57 2bf0             sub     esi,eax
bf920e59 d1fe             sar     esi,1
```

!cppexr

8/26/2019 • 2 minutes to read • Edit Online

The **!cppexr** extension displays the contents of a C++ exception record.

```
!cppexr Address
```

Parameters

Address
Specifies the address of the C++ exception record to display.

DLL

Windows 2000	Ext.dll
Windows XP and later	Ext.dll

Additional Information

For more information about exceptions, see Controlling Exceptions and Events, the Windows Driver Kit (WDK) documentation, the Windows SDK documentation, and *Microsoft Windows Internals* by Mark Russinovich and David Solomon. Use the **.exr** command to display other exception records.

Remarks

The **!cppexr** extension displays information that is related to a C++ exception that the target encounters, including the exception code, the address of the exception, and the exception flags. This exception must be one of the standard C++ exceptions that are defined in Msvcrt.dll.

You can typically obtain the *Address* parameter by using the **!analyze -v** command.

The **!cppexr** extension is useful for determining the type of a C++ exception.

!cpuid

12/5/2018 • 2 minutes to read • Edit Online

The **!cpuid** extension displays information about the processors on the system.

```
!cpuid [Processor]
```

Parameters

Processor
Specifies the processor whose information will be displayed. If you omit this parameter, all processors are displayed.

DLL

Windows 2000	Ext.dll
Windows XP and later	Ext.dll

Additional Information

For more information about how to debug multiprocessor computers, see Multiprocessor Syntax.

Remarks

The **!cpuid** extension works during live user-mode or kernel-mode debugging, local kernel debugging, and debugging of dump files. However, user-mode minidump files contain only information about the active processor.

If you are debugging in user mode, the **!cpuid** extension describes the computer that the target application is running on. In kernel mode, it describes the target computer.

The following example shows this extension.

```
kd> !cpuid
CP  F/M/S  Manufacturer        MHz
 0  6,5,1  GenuineIntel        700
 1  8,1,5  AuthenticAMD        700
```

The **CP** column gives the processor number. (These numbers are always sequential, starting with zero). The **Manufacturer** column specifies the processor manufacturer. The **MHz** column specifies the processor speed, if it is available.

For an x86-based processor or an x64-based processor, the **F** column displays the processor family number, the **M** column displays the processor model number, and the **S** column displays the stepping size.

For an Itanium-based processor, the **M** column displays the processor model number, the R column displays the processor revision number, the **F** column displays the processor family number, and the **A** column displays the architecture revision number.

!cs

6/7/2019 • 3 minutes to read • Edit Online

The **!cs** extension displays one or more critical sections or the whole critical section tree.

```
!cs [-s] [-l] [-o]
!cs [-s] [-o] Address
!cs [-s] [-l] [-o] StartAddress EndAddress
!cs [-s] [-o] -d InfoAddress
!cs [-s] -t [TreeAddress]
!cs -?
```

Parameters

PARAMETER	DESCRIPTION
-s	Displays each critical section's initialization stack trace, if this information is available.
-l	Display only the locked critical sections.
-o	Displays the owner's stack for any locked critical section that is being displayed.
Address	Specifies the address of the critical section to display. If you omit this parameter, the debugger displays all critical sections in the current process.
StartAddress	Specifies the beginning of the address range to search for critical sections.
EndAddress	Specifies the end of the address range to search for critical sections.
-d	Displays critical sections that are associated with DebugInfo.
InfoAddress	Specifies the address of the DebugInfo.
-t	Displays a critical section tree. Before you can use the **-t** option, you must activate Application Verifier for the target process and select the **Check lock usage** option.
TreeAddress	Specifies the address of the root of the critical section tree. If you omit this parameter or specify zero, the debugger displays the critical section tree for the current process.
-?	Displays some Help text for this extension in the Debugger Command window.

DLL

Exts.dll

Additional Information

For other commands and extensions that can display critical section information, see Displaying a Critical Section. For more information about critical sections, see the Microsoft Windows SDK documentation, the Windows Driver Kit (WDK) documentation, and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

The **!cs** extension requires full symbols (including type information) for the process that is being debugged and for Ntdll.dll. If you do not have symbols for Ntdll.dll, see Installing Windows Symbol Files.

The following examples shows you how to use **!cs**. The following command displays information about the critical section at address 0x7803B0F8 and shows its initialization stack trace.

```
0:001> !cs -s 0x7803B0F8
Critical section   = 0x7803B0F8 (MSVCRT!__app_type+0x4)
DebugInfo          = 0x6A262080
NOT LOCKED
LockSemaphore      = 0x0
SpinCount          = 0x0

Stack trace for DebugInfo = 0x6A262080:

0x6A2137BD: ntdll!RtlInitializeCriticalSectionAndSpinCount+0x9B
0x6A207A4C: ntdll!LdrpCallInitRoutine+0x14
0x6A205569: ntdll!LdrpRunInitializeRoutines+0x1D9
0x6A20DCE1: ntdll!LdrpInitializeProcess+0xAE5
```

The following command displays information about the critical section whose DebugInfo is at address 0x6A262080.

```
0:001> !cs -d 0x6A262080
DebugInfo          = 0x6A262080
Critical section   = 0x7803B0F8 (MSVCRT!__app_type+0x4)
NOT LOCKED
LockSemaphore      = 0x0
SpinCount          = 0x0
```

The following command displays information about all of the active critical sections in the current process.

```
## 0:001> !cs

DebugInfo          = 0x6A261D60
Critical section   = 0x6A262820 (ntdll!RtlCriticalSectionLock+0x0)
LOCKED
LockCount          = 0x0
OwningThread       = 0x460
RecursionCount     = 0x1
LockSemaphore      = 0x0
## SpinCount          = 0x0

DebugInfo          = 0x6A261D80
Critical section   = 0x6A262580 (ntdll!DeferedCriticalSection+0x0)
NOT LOCKED
LockSemaphore      = 0x7FC
## SpinCount          = 0x0

DebugInfo          = 0x6A262600
Critical section   = 0x6A26074C (ntdll!LoaderLock+0x0)
NOT LOCKED
LockSemaphore      = 0x0
## SpinCount          = 0x0

DebugInfo          = 0x77fbde20
Critical section   = 0x77c8ba60 (GDI32!semColorSpaceCache+0x0)
LOCKED
LockCount          = 0x0
OwningThread       = 0x00000dd8
RecursionCount     = 0x1
LockSemaphore      = 0x0
## SpinCount          = 0x00000000

...
```

The following command displays the critical section tree.

```
0:001> !cs -t

Tree root 00bb08c0

Level     Node       CS    Debug  InitThr EnterThr  WaitThr TryEnThr LeaveThr EnterCnt  WaitCnt
##


    0 00bb08c0 77c7e020 77fbcae0      4c8      4c8        0        0      4c8        c        0
 1 00dd6fd0 0148cfe8 01683fe0      4c8      4c8        0        0      4c8        2        0
 2 00bb0aa0 008e8b84 77fbcc20      4c8        0        0        0        0        0        0
    3 00bb09e0 008e8704 77fbcba0      4c8        0        0        0        0        0        0
    4 00bb0a40 008e8944 77fbcbe0      4c8        0        0        0        0        0        0
    5 00bb0a10 008e8824 77fbcbc0      4c8        0        0        0        0        0        0
    5 00bb0a70 008e8a64 77fbcc00      4c8        0        0        0        0        0        0
    3 00bb0b00 008e8dc4 77fbcc60      4c8        0        0        0        0        0        0
    4 00bb0ad0 008e8ca4 77fbcc40      4c8        0        0        0        0        0        0
    4 00bb0b30 008e8ee4 77fbcc80      4c8        0        0        0        0        0        0
    5 00dd4fd0 0148afe4 0167ffe0      4c8        0        0        0        0        0        0
    2 00bb0e90 77c2da98 00908fe0      4c8      4c8        0        0      4c8       3a        0
 3 00bb0d70 77c2da08 008fcfe0      4c8        0        0        0        0        0        0
```

The following items appear in this **!cs -t** display:

- **InitThr** is the thread ID for the thread that initialized the CS.
- **EnterThr** is the ID of the thread that called **EnterCriticalSection** last time.

- **WaitThr** is the ID of the thread that found the critical section that another thread owned and waited for it last time.
- **TryEnThr** is the ID of the thread that called **TryEnterCriticalSection** last time.
- **LeaveThr** is the ID of the thread that called **LeaveCriticalSection** last time
- **EnterCnt** is the count of **EnterCriticalSection**.
- **WaitCnt** is the contention count.

!cxr

12/5/2018 • 2 minutes to read • Edit Online

The **!cxr** extension command is obsolete. Use the **.cxr (Display Context Record)** command instead.

!dh

12/5/2018 • 2 minutes to read • Edit Online

The **!dh** extension displays the headers for the specified image.

```
!dh [Options] Address
!dh -h
```

Parameters

Options
Any one of the following options:

-f
Displays file headers.

-s
Displays section headers.

-a
Displays all header information.

Address
Specifies the hexadecimal address of the image.

-h
Displays some Help text for this extension in the Debugger Command window.

DLL

Windows 2000	Dbghelp.dll Kdextx86.dll Ntsdexts.dll
Windows XP and later	Dbghelp.dll

Remarks

The **!lmi** extension extracts the most important information from the image header and displays it in a concise summary format. That extension is frequently more useful than **!dh**.

!dlls

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

The **!dlls** extension displays the table entries of all loaded modules or all modules that a specified thread or process are using.

```
!dlls [Options] [LoaderEntryAddress]
!dlls -h
```

Parameters

Options
Specifies the level of output. This parameter can be any combination of the following values:

-f
Displays file headers.

-s
Displays section headers.

-a
Displays complete module information. (This option is equivalent to -f -s.)

-c **** *ModuleAddress*
Displays the module that contains *ModuleAddress*.

-i
Sorts the display by initialization order.

-l
Sorts the display by load order. This situation is the default.

-m
Sorts the display by memory order.

-v
Displays version information. This information is taken from the resource section of each module.

LoaderEntryAddress
Specifies the address of the loader entry for a module. If you include this parameter, the debugger displays only this specific module.

-h
Displays some Help text for this extension in the Debugger Command window.

DLL

Windows 2000	Kdextx86.dll Ntsdexts.dll
Windows XP and later	Exts.dll

Remarks

The module listing includes all entry points into each module.

The **.dlls** extension works only in live debugging (not with crash dump analysis).

In kernel mode, this extension displays the modules for the current process context. You cannot use **!dlls** together with a system process or the idle process.

The following examples shows you how to use the **!dlls** extension.

```
kd> !dlls -c 77f60000
Dump dll containing 0x77f60000:

0x00091f38: E:\WINDOWS\System32\ntdll.dll
      Base   0x77f60000  EntryPoint  0x00000000  Size        0x00097000
      Flags  0x00004004  LoadCount   0x0000ffff  TlsIndex    0x00000000
             LDRP_IMAGE_DLL
             LDRP_ENTRY_PROCESSED

kd> !dlls -a 91ec0

0x00091ec0: E:\WINDOWS\system32\winmine.exe
      Base   0x01000000  EntryPoint  0x01003e2e  Size        0x00020000
      Flags  0x00005000  LoadCount   0x0000ffff  TlsIndex    0x00000000
             LDRP_LOAD_IN_PROGRESS
             LDRP_ENTRY_PROCESSED

File Type: EXECUTABLE IMAGE
FILE HEADER VALUES
     14C machine (i386)
       3 number of sections
3A98E856 time date stamp Sun Feb 25 03:11:18 2001

       0 file pointer to symbol table
       0 number of symbols
      E0 size of optional header
     10F characteristics
            Relocations stripped
            Executable
            Line numbers stripped
            Symbols stripped
            32 bit word machine

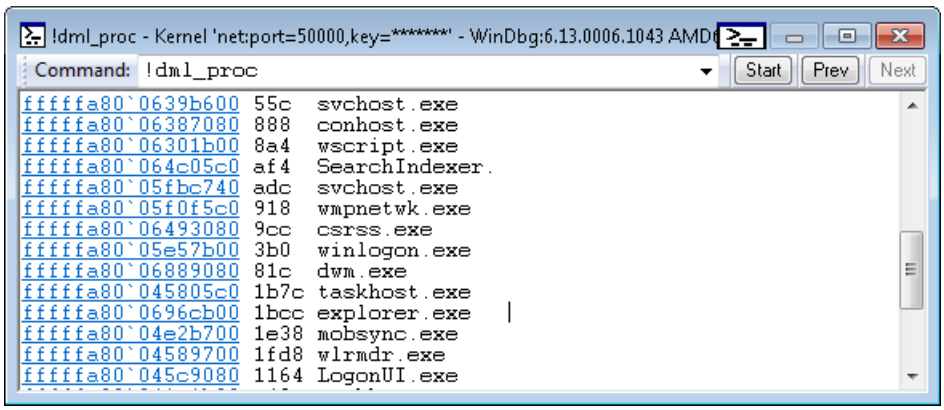
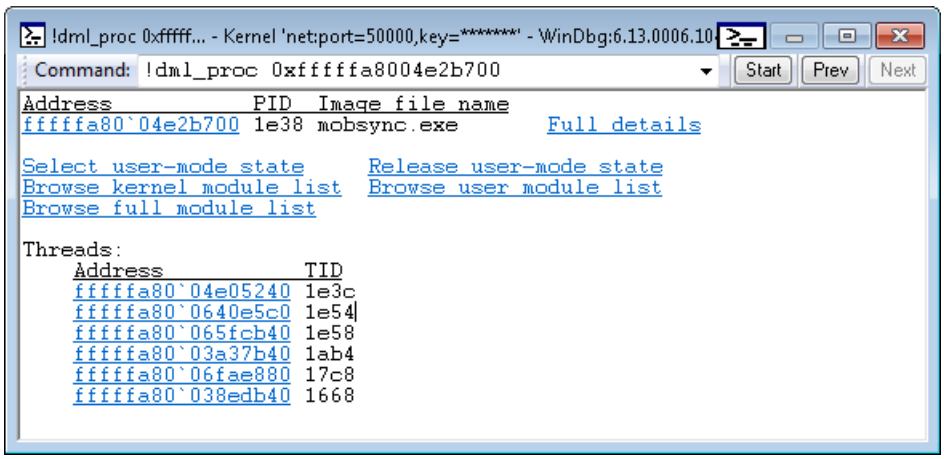
OPTIONAL HEADER VALUES
     10B magic #
    7.00 linker version
    3A00 size of code
   19E00 size of initialized data
       0 size of uninitialized data
    3E2E address of entry point
    1000 base of code
         ----- new -----
01000000 image base
    1000 section alignment
     200 file alignment
       2 subsystem (Windows GUI)
    5.01 operating system version
    5.01 image version
    4.00 subsystem version
   20000 size of image
     400 size of headers
   21970 checksum
00040000 size of stack reserve
00001000 size of stack commit
00100000 size of heap reserve
00001000 size of heap commit
```

```
01000100 Opt Hdr
       0 [       0] address [size] of Export Directory
    40B4 [      B4] address [size] of Import Directory
    6000 [   19170] address [size] of Resource Directory
       0 [       0] address [size] of Exception Directory
       0 [       0] address [size] of Security Directory
       0 [       0] address [size] of Base Relocation Directory
    11B0 [      1C] address [size] of Debug Directory
       0 [       0] address [size] of Description Directory
       0 [       0] address [size] of Special Directory
       0 [       0] address [size] of Thread Storage Directory
       0 [       0] address [size] of Load Configuration Directory
     258 [      A8] address [size] of Bound Import Directory
    1000 [     1B0] address [size] of Import Address Table Directory
       0 [       0] address [size] of Reserved Directory
       0 [       0] address [size] of Reserved Directory
       0 [       0] address [size] of Reserved Directory


SECTION HEADER #1
   .text name
    3992 virtual size
    1000 virtual address
    3A00 size of raw data
     400 file pointer to raw data
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
60000020 flags
         Code
         (no align specified)
         Execute Read


Debug Directories(1)
        Type       Size     Address  Pointer

        cv           1c        13d0      7d0    Format: NB10, 3a98e856, 1, winmi
ne.pdb

SECTION HEADER #2
   .data name
     BB8 virtual size
    5000 virtual address
     200 size of raw data
    3E00 file pointer to raw data
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
C0000040 flags
         Initialized Data
         (no align specified)
         Read Write

SECTION HEADER #3
   .rsrc name
   19170 virtual size
    6000 virtual address
   19200 size of raw data
    4000 file pointer to raw data
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
40000040 flags
         Initialized Data
         (no align specified)
```

```
(no align specified)
Read Only
```

!dml_proc

12/5/2018 • 2 minutes to read • Edit Online

The **!dml_proc** extension displays a list of processes and provides links for obtaining more detailed information about processes.

```
!dml_proc
```

Remarks

The following image shows a portion of the output displayed by **!dml_proc**.



In the preceding output, the process addresses are links that you can click to see more detailed information. For example, if you click **fffffa80`04e2b700** (the address for mobsync.exe), you will see detailed information about the mobsync.exe process as shown in the following image.



The preceding output, which describes an individual process, contains links that you can click to explore the process and its threads in more detail.

See also

Debugger Markup Language Commands

!dumpfa

12/5/2018 • 2 minutes to read • Edit Online

The **!dumpfa** extension displays the contents of a failure analysis entry.

```
!dumpfa Address
```

Parameters

Address
Specifies the address of the failure analysis entry that is displayed.

DLL

Windows 2000	Ext.dll
Windows XP and later	Ext.dll

Remarks

The **.dumpfa** extension is useful only to debug the **!analyze** extension, as the following example shows.

```
0:000> !dumpfa 0x00a34140
DataUsed 3b0
Type =        DEBUG_FLR_MARKER_BUCKET 00010016 - Size = 9
Type =          DEBUG_FLR_MARKER_FILE 0001000d - Size = 16
Type =      DEBUG_FLR_SYSXML_LOCALEID 00004200 - Size = 4
Type =      DEBUG_FLR_SYSXML_CHECKSUM 00004201 - Size = 4
Type =         DEBUG_FLR_READ_ADDRESS 0000000e - Size = 8
Type =          DEBUG_FLR_FAULTING_IP 80000000 - Size = 8
Type =     DEBUG_FLR_MM_INTERNAL_CODE 00001004 - Size = 8
Type = DEBUG_FLR_CPU_MICROCODE_VERSION 0000301f - Size = 28
Type = DEBUG_FLR_CUSTOMER_CRASH_COUNT 0000300b - Size = 8
Type =    DEBUG_FLR_DEFAULT_BUCKET_ID 00010008 - Size = 12
Type =         DEBUG_FLR_BUGCHECK_STR 00000600 - Size = 5
Type = DEBUG_FLR_LAST_CONTROL_TRANSFER 0000000a - Size = 18
Type =           DEBUG_FLR_TRAP_FRAME c0000002 - Size = 8
Type =           DEBUG_FLR_STACK_TEXT 00010005 - Size = 1fb
Type =        DEBUG_FLR_STACK_COMMAND 00010004 - Size = 17
Type =        DEBUG_FLR_OS_BUILD_NAME 0000301e - Size = 9
Type =          DEBUG_FLR_MODULE_NAME 00010006 - Size = 8
Type =           DEBUG_FLR_IMAGE_NAME 00010001 - Size = c
Type =      DEBUG_FLR_IMAGE_TIMESTAMP 80000002 - Size = 8
```

You can also use the **!asd** extension to debug the **!analyze** extension.

2 minutes to read

!envvar

12/5/2018 • 2 minutes to read • Edit Online

The **!envvar** extension displays the value of the specified environment variable.

```
!envvar Variable
```

Parameters

Variable
Specifies the environment variable whose value is displayed. *Variable* is not case sensitive.

DLL

Windows 2000	Unavailable
Windows XP and later	Exts.dll

Additional Information

For more information about environment variables, see Environment Variables and the Microsoft Windows SDK documentation.

Remarks

The **!envvar** extension works both in user mode and in kernel mode. However, in kernel mode, when you set the idle thread as the current process, the pointer to the Process Environment Block (PEB) is **NULL**, so it fails. In kernel mode, the **!envvar** extension displays the environment variables on the target computer, as the following example shows.

```
0:000> !envvar _nt_symbol_path
        _nt_symbol_path = srv*C:\mysyms*https://msdl.microsoft.com/download/symbols
```

!error

12/5/2018 • 2 minutes to read • Edit Online

The **!error** extension decodes and displays information about an error value.

```
!error Value [Flags]
```

Parameters

Value
Specifies one of the following error codes:

- Win32
- Winsock
- NTSTATUS
- NetAPI

Flags
If *Flags* is set to 1, the error code is read as an NTSTATUS code.

DLL

Windows 2000	Ext.dll
Windows XP and later	Ext.dll

Remarks

The following example shows you how to use **!error**.

```
0:000> !error 2
Error code: (Win32) 0x2 (2) - The system cannot find the file specified.
0:000> !error 2 1
Error code: (NTSTATUS) 0x2 - STATUS_WAIT_2
```

!exchain

12/5/2018 • 2 minutes to read • Edit Online

The **!exchain** extension displays the current exception handler chain.

```
!exchain [Options]
```

Parameters

Options
One of the following values:

/c
Displays information that is relevant for debugging a C++ **try**/**catch** exception, if such an exception is detected.

/C
Displays information that is relevant for debugging a C++ **try**/**catch** exception, even when such an exception has not been detected.

/f
Displays information that is obtained by walking the CRT function tables, even if a CRT exception handler was not detected.

DLL

Windows 2000	Ext.dll
Windows XP and later	Ext.dll

The **!exchain** extension is available only for an x86-based target computer.

Remarks

The **!exchain** extension displays the list of exception handlers for the current thread.

The list begins with the first handler on the chain (the one that is given the first opportunity to handle an exception) and continues on to the end. The following example shows this extension.

```
0:000> !exchain
0012fea8: Prymes!_except_handler3+0 (00407604)
  CRT scope  0, filter: Prymes!dzExcepError+e6 (00401576)
                func:   Prymes!dzExcepError+ec (0040157c)
0012ffb0: Prymes!_except_handler3+0 (00407604)
  CRT scope  0, filter: Prymes!mainCRTStartup+f8 (004021b8)
                func:   Prymes!mainCRTStartup+113 (004021d3)
0012ffe0: KERNEL32!GetThreadContext+1c (77ea1856)
```

!exr

12/5/2018 • 2 minutes to read • Edit Online

The **!exr** extension command is obsolete. Use the **.exr (Display Exception Record)** command instead.

!findxmldata

12/5/2018 • 2 minutes to read • Edit Online

The **!findxmldata** extension retrieves XML data from a CAB file that contains a kernel-mode Small Memory Dump file.

```
!findxmldata [ -d DeviceName | -h HwId ]
!findxmldata -r Driver
!findxmldata -chksum [ -z CabFile ]
!findxmldata -v
```

Parameters

-d *DeviceName*
Displays all devices whose device name contains the string that *DeviceName* specifies.

-h *HwId*
Displays all devices whose hardware IDs contain the string that *HwId* specifies. If you use both **-d** and **-h**, the debugger displays only those devices that satisfy both matches.

-r *Driver*
Displays information about the driver that the *Driver* parameter specifies, including all devices that use this driver.

-chksum
Displays the XML file's checksum.

-z *CabFile*
Enables you to perform a checksum on the CAB file that the CabFile parameter specifies, instead of on the default Sysdata.xml file.

-v
Displays system version information.

DLL

Windows 2000	Ext.dll
Windows XP and later	Ext.dll

The **!findxmldata** extension works only on a kernel-mode Small Memory Dump file that is stored in a CAB file.

Additional Information

For more information about how to put dump files into CAB files, see **.dumpcab (Create Dump File CAB)**. For information more about how to debug a kernel-mode dump file, including dump files that are stored inside CAB files, see Analyzing a Kernel-Mode Dump File.

Remarks

The **!findxmldata** extension retrieves data from the Sysdata.xml file that is stored in a CAB file that contains a kernel-mode Small Memory Dump file.

When you do not use any options, the extension displays all devices.

The following examples show you how to use **!findxmldata**.

```
kd> !findxmldata -v
SYSTEM Info:
OSVER: 5.1.2600 2.0
OSLANGUAGE: 2052
OSNAME: Microsoft Windows XP Home Edition
kd> !findxmldata -d MIDI
Node DEVICE
 DESCRIPTION    : MPU-401 Compatible MIDI Device
        HARDWAREID     : ACPI\PNPB006
        SERVICE        : ms_mpu401
        DRIVER         : msmpu401.sys

kd> !findxmldata -r msmpu
Node DRIVER
 FILENAME       : msmpu401.sys
        FILESIZE       : 2944
        CREATIONDATE   : 05-06-2005 09:18:34
        VERSION        : 5.1.2600.0
        MANUFACTURER   : Microsoft Corporation
        PRODUCTNAME    : Microsoft« Windows« Operating System
Node DEVICE
        DESCRIPTION    : MPU-401 Compatible MIDI Device
 HARDWAREID     : ACPI\PNPB006
        SERVICE        : ms_mpu401
        DRIVER         : msmpu401.sys

kd> !findxmldata -h PCI\VEN_8086&DEV_24C3&SUBSYS_24C28086
Node DEVICE
 DESCRIPTION    : Intel(R) 82801DB/DBM SMBus Controller - 24C3
        HARDWAREID     : PCI\VEN_8086&DEV_24C3&SUBSYS_24C28086&REV_01
kd> !findxmldata -h USB\ROOT_HUB&VID8086&PID24C4&REV0001
Node DEVICE
        DESCRIPTION    : USB Root Hub
 HARDWAREID     : USB\ROOT_HUB&VID8086&PID24C4&REV0001
        SERVICE        : usbhub
        DRIVER         : usbhub.sys

kd> !findxmldata -h ACPI\PNPB006
Node DEVICE
        DESCRIPTION    : MPU-401 Compatible MIDI Device
 HARDWAREID     : ACPI\PNPB006
        SERVICE        : ms_mpu401
        DRIVER         : msmpu401.sys
```

!for_each_frame

12/5/2018 • 2 minutes to read • Edit Online

The **!for_each_frame** extension executes a debugger command one time for each frame in the stack of the current thread.

```
!for_each_frame ["CommandString"]
!for_each_frame -?
```

Parameters

CommandString
Specifies the debugger commands to execute one time for each frame. If *CommandString* includes multiple commands, you must separate them with semicolons and enclose *CommandString* in quotation marks. If you include multiple commands, the individual commands within *CommandString* cannot contain quotation marks. If you want to refer to the index of the current frame within *CommandString*, use the @$frame pseudoregister.

-?
Displays some Help text for this extension in the Debugger Command window.

DLL

Windows 2000	Ext.dll
Windows XP and later	Ext.dll

Additional Information

For more information about the local context, see Changing Contexts.

Remarks

If you do not specify any arguments, the **!for_each_frame** extension displays a list of all frames and their frame indexes. For a more detailed list of all frames, use the **k (Display Stack Backtrace)** command.

The **k** command walks up to 256 frames. For each enumerated frame, that frame temporarily becomes the current local context (similar to the **.frame (Set Local Context)** command). After the context has been set, *CommandString* is executed. After all frames have been used, the local context is reset to the value that it had before you used the **!for_each_frame** extension.

If you include *CommandString*, the debugger displays the frame and its index before the command is executed for that frame.

The following command displays all local variables for the current stack.

```
!for_each_frame !for_each_local dt @#Local
```

!for_each_function

12/5/2018 • 2 minutes to read • Edit Online

The **!for_each_function** extension executes a debugger command for each function, in a specified module, whose name matches a specified pattern.

```
!for_each_function -m:ModuleName -p:Pattern -c:CommandString
!for_each_function -?
```

Parameters

-m:*ModuleName*
Specifies the module name. This name is typically the file name without the file name extension. In some cases, the module name differs significantly from the file name.

-p:*Pattern*
Specifies the pattern to be matched.

-c:*CommandString*
Specifies the debugger command to execute for each function, in the specified module, that matches the pattern.

You can use the following aliases in *CommandString*.

ALIAS	DATA TYPE	VALUE
@#SymbolName	string	The symbol name.
@#SymbolAddress	ULONG64	The symbol address.
@#ModName	string	The module name.
@#FunctionName	string	The function name.

-?
Displays help for this extension.

DLL

Ext.dll

Remarks

The following example shows how to list all function names, in the PCI module, that match the pattern *read*.

```
1: kd> !for_each_function -m:pci -p:*read* -c:.echo @#FunctionName

PciReadDeviceConfig
PciReadDeviceSpace
PciReadSlotIdData
PciExternalReadDeviceConfig
PiRegStateReadStackCreationSettingsFromKey
VmProxyReadDevicePathsFromRegistry
PpRegStateReadCreateClassCreationSettings
ExpressRootPortReadConfigSpace
PciReadRomImage
PciDevice_ReadConfig
PciReadDeviceConfigEx
PciReadSlotConfig
```

If an alias is not preceded and followed by a blank space, you must put the alias inside the **${} Alias Interpreter** token.

The following example shows how to list all symbols, in all modules, whose function names match the pattern *CreateFile*. The alias @#ModuleName is not preceded by a blank space. Therefore, it is put inside the **${} Alias Interpreter** token.

Note Do not confuse the @#ModuleName alias with the @#ModName alias. The @#ModuleName alias belongs to the **!for_each_module** extension, and the @#ModName alias belongs to the **!for_each_function** extension.

```
1: kd> !for_each_module !for_each_function -m:${@#ModuleName} -p:*CreateFile* -c:.echo @#SymbolName
nt!BiCreateFileDeviceElement
nt!NtCreateFile
...
Wdf01000!FxFileObject::_CreateFileObject
fltmgr!FltCreateFileEx2$fin$0
fltmgr!FltCreateFileEx2
...
Ntfs!TxfIoCreateFile
Ntfs!NtfsCreateFileLock
...
MpFilter!MpTxfpCreateFileEntryUnsafe
mrxsmb10!MRxSmbFinishLongNameCreateFile
...
srv!SrvIoCreateFile
```

You can put a sequence of commands in a command file, and use **$$>< (Run Script File)** to execute those commands for each function that matches the pattern. Suppose that a file named Commands.txt contains the following commands:

```
.echo
.echo @#FunctionName
u @#SymbolAddress L1
```

In the following example, the commands in the Commands.text file are executed for each function, in the PCI module, that matches the pattern *read*.

```
1: kd> !for_each_function -m:pci -p:*read* -c:$$><Commands.txt

PciReadDeviceConfig
pci!PciReadDeviceConfig [d:\wmm1\drivers\busdrv\pci\config.c @ 349]:
fffff880`00f7b798 48895c2408      mov     qword ptr [rsp+8],rbx

PciReadDeviceSpace
pci!PciReadDeviceSpace [d:\wmm1\drivers\busdrv\pci\config.c @ 1621]:
fffff880`00f7c044 48895c2408      mov     qword ptr [rsp+8],rbx

...
```

See also

!for_each_module

!for_each_local

12/5/2018 • 2 minutes to read • Edit Online

The **!for_each_local** extension executes a debugger command one time for each local variable in the current frame.

```
!for_each_local ["CommandString"]
!for_each_local -?
```

Parameters

CommandString
Specifies the debugger commands to execute one time for each local variable in the current stack frame. If *CommandString* includes multiple commands, you must separate them with semicolons and enclose *CommandString* in quotation marks. If you include multiple commands, the individual commands in *CommandString* cannot contain quotation marks.

Within *CommandString*, or within any script that the commands in *CommandString* execute, you can use the **@#Local** alias. This alias is replaced by the name of the local variable. This replacement occurs before *CommandString* is executed and before any other parsing occurs. This alias is case sensitive, and you must add a space before it and add a space after it, even if you enclose the alias in parentheses. If you use C++ expression syntax, you must reference this alias as **@@(@#Local)**.

This alias is available only during the lifetime of the call to **!for_each_local**. Do not confuse this alias with pseudo-registers, fixed-name aliases, or user-named aliases.

-?
Displays some Help text for this extension in the Debugger Command window.

DLL

Windows 2000	Ext.dll
Windows XP and later	Ext.dll

Additional Information

For more information about how to display and change local variables and a description of other memory-related commands, see Reading and Writing Memory.

Remarks

If you do not specify any arguments, the **!for_each_local** extension lists local variables. For more information about the local variables, use the **dv (Display Local Variables)** command.

If you enable verbose debugger output, the display includes the total number of local variables when the extension is called, and every time that *CommandString* is executed for a local variable, that variable and the text of *CommandString* are echoed.

!for_each_module

12/5/2018 • 4 minutes to read • Edit Online

The **!for_each_module** extension executes a debugger command one time for each loaded module.

```
!for_each_module ["CommandString"]
!for_each_module -?
```

Parameters

CommandString
Specifies the debugger commands to execute one time for each module in the debugger's module list. If *CommandString* includes multiple commands, you must separate them with semicolons and enclose *CommandString* in quotation marks. If you include multiple commands, the individual commands within *CommandString* cannot contain quotation marks.

You can use the following aliases in *CommandString* or in any script that the commands in *CommandString* executes.

ALIAS	DATA TYPE	VALUE
@#FileVersion	string	The file version of the module.
@#ProductVersion	string	The product version of the module.
@#ModuleIndex	ULONG	The module number. Modules are enumerated consecutively, starting with zero.
@#ModuleName	string	The module name. This name is typically the file name without the file name extension. In some situations, the module name differs significantly from the file name.
@#ImageName	string	The name of the executable file, including the file name extension. Typically, the full path is included in user mode but not in kernel mode.
@#LoadedImageName	string	Unless Microsoft CodeView symbols are present, this alias is the same as the image name.

ALIAS	DATA TYPE	VALUE
@#MappedImageName	string	In most situations, this alias is **NULL**. If the debugger is mapping an image file (for example, during minidump debugging), this alias is the name of the mapped image.
@#SymbolFileName	string	The path and name of the symbol file. If you have not loaded any symbols, this alias is the name of the executable file instead.
@#ModuleNameSize	ULONG	The string length of the module name string, plus one.
@#ImageNameSize	ULONG	The string length of the image name string, plus one.
@#LoadedImageNameSize	ULONG	The string length of the loaded image name string, plus one.
@#MappedImageNameSize	ULONG	The string length of the mapped image name string, plus one.
@#SymbolFileNameSize	ULONG	The string length of the symbol file name string, plus one.
@#Base	ULONG64	The address of the start of the image.
@#Size	ULONG	The size of the image, in bytes.
@#End	ULONG64	The address of the end of the image.
@#TimeDateStamp	ULONG	The time and date stamp of the image. If you want to expand this time and date stamp into a readable date, use the **.formats (Show Number Formats)** command.
@#Checksum	ULONG	The checksum of the module.

ALIAS	DATA TYPE	VALUE
@#Flags	ULONG	The module flags. For a list of the DEBUG_MODULE_*Xxx* values, see Dbgeng.h.
@#SymbolType	USHORT	The symbol type. For a list of the DEBUG_SYMTYPE_*Xxx* values, see Dbgeng.h.

These aliases are all replaced before *CommandString* is executed for each module and before any other parsing occurs. These aliases are case sensitive. You must add a space before the alias and a space after it, even if the alias is enclosed in parentheses. If you use C++ expression syntax, you must reference these aliases as @@(@#*alias*).

These aliases are available only during the lifetime of the call to **!for_each_module**. Do not confuse them with pseudo-registers, fixed-name aliases, or user-named aliases.

-?
Displays some Help text for this extension in the Debugger Command window.

DLL

Windows 2000	Ext.dll
Windows XP and later	Ext.dll

Additional Information

For more information about how to define and use aliases as shortcuts for entering character strings (including use of the ${ } token), see Using Aliases.

Remarks

If you do not specify any arguments, the **!for_each_module** extension displays general information about the loaded modules. This information is similar to the information that the following command shows.

```
!for_each_module .echo @#ModuleIndex : @#Base @#End @#ModuleName @#ImageName  @#LoadedImageName
```

For more information about loaded and unloaded modules, use the **lm (List Loaded Modules)** command.

If you enable verbose debugger output, the debugger displays the total number of loaded and unloaded modules when the extension is called, and the debugger displays detailed information about each module (including the values of each available alias) before *CommandString* is executed for that module.

The following examples show how to use the **!for_each_module** extension. The following commands display the global debug flags.

```
!for_each_module x ${@#ModuleName}!*Debug*Flag*
!for_each_module x ${@#ModuleName}!g*Debug*
```

The following command checks for binary corruption in every loaded module, by using the **!chkimg** extension:

```
!for_each_module !chkimg @#ModuleName
```

The following command searches for the pattern "MZ" in every loaded image.

```
!for_each_module s-a @#Base @#End "MZ"
```

The following example demonstrates the use of @#FileVersion and @#ProductVersion for each module name:

```
0:000> !for_each_module .echo @#ModuleName fver = @#FileVersion pver = @#ProductVersion
USER32 fver = 6.0.6000.16438 (vista_gdr.070214-1610) pver = 6.0.6000.16438
kernel32 fver = 6.0.6000.16386 (vista_rtm.061101-2205) pver = 6.0.6000.16386
ntdll fver = 6.0.6000.16386 (vista_rtm.061101-2205) pver = 6.0.6000.16386
notepad fver = 6.0.6000.16386 (vista_rtm.061101-2205) pver = 6.0.6000.16386
WINSPOOL fver = 6.0.6000.16386 (vista_rtm.061101-2205) pver = 6.0.6000.16386
COMCTL32 fver = 6.10 (vista_rtm.061101-2205) pver = 6.0.6000.16386
SHLWAPI fver = 6.0.6000.16386 (vista_rtm.061101-2205) pver = 6.0.6000.16386
msvcrt fver = 7.0.6000.16386 (vista_rtm.061101-2205) pver = 7.0.6000.16386
GDI32 fver = 6.0.6000.16386 (vista_rtm.061101-2205) pver = 6.0.6000.16386
RPCRT4 fver = 6.0.6000.16525 (vista_gdr.070716-1600) pver = 6.0.6000.16525
SHELL32 fver = 6.0.6000.16513 (vista_gdr.070626-1505) pver = 6.0.6000.16513
ole32 fver = 6.0.6000.16386 (vista_rtm.061101-2205) pver = 6.0.6000.16386
ADVAPI32 fver = 6.0.6000.16386 (vista_rtm.061101-2205) pver = 6.0.6000.16386
COMDLG32 fver = 6.0.6000.16386 (vista_rtm.061101-2205) pver = 6.0.6000.16386
```

!for_each_register

12/5/2018 • 2 minutes to read • Edit Online

The **!for_each_register** extension executes a specified command for each register.

```
!for_each_register -c:CommandString
!for_each_register -?
```

Parameters

-c:*CommandString*
Specifies the command to be executed for each register. The aliases @#RegisterName and @#RegisterValue are valid during the execution of the command.

-?
Displays help for the **!for_each_register** extension.

DLL

Ext.dll

Examples

This example lists the name of each register.

```
0:000> !for_each_register -c:.echo @#RegisterName
rax
rcx
rdx
rbx
...
```

This example executes **!address** for each register value.

```
0:000> !for_each_register -c:!address ${@#RegisterValue}
...
Usage:                  Stack
Base Address:           00000008`a568f000
End Address:            00000008`a56a0000
Region Size:            00000000`00011000
State:                  00001000    MEM_COMMIT
Protect:                00000004    PAGE_READWRITE
Type:                   00020000    MEM_PRIVATE
Allocation Base:        00000008`a5620000
Allocation Protect:     00000004    PAGE_READWRITE
More info:              ~0k
...
```

Remarks

When an alias is an argument to a debugger extension (for example, **!address**), use the alias interpreter **${} (Alias Interpreter)** token so that the alias is resolved correctly.

For more information about how to define and use aliases as shortcuts for entering character strings (including use of the **${}** token), see Using Aliases.

!gflag

3/6/2019 • 2 minutes to read • Edit Online

The **!gflag** extension sets or displays the global flags.

```
!gflag [+|-] Value
!gflag {+|-} Abbreviation
!gflag -?
!gflag
```

Parameters

Value
Specifies a 32-bit hexadecimal number. If you do not use a plus sign (**+**) or minus sign (**-**), this number becomes the new value of the global flag bit field. If you add a plus sign (**+**) before this number, the number specifies one or more global flag bits to set to 1. If you add a minus sign (**-**) before this number, the number specifies one or more global flag bits to set to zero.

Abbreviation
Specifies a single global flag. *Abbreviation* is a three-letter abbreviation for a global flag that is set to 1 (**+**) or to zero (**-**).

-?
Displays some Help text for this extension, including a list of global flag abbreviations, in the Debugger Command window.

DLL

Windows 2000	Kdextx86.dll Ntsdexts.dll
Windows XP and later	Exts.dll

Additional Information

You can also set these flags by using the Global Flags utility (Gflags.exe).

Remarks

If you do not specify any arguments, the **!gflag** extension displays the current global flag settings.

The following table contains the abbreviations that you can use for the *Abbreviation* parameter.

VALUE	NAME	DESCRIPTION
0x00000001	"soe"	Stop on exception.
0x00000002	"sls"	Show loader snaps.

VALUE	NAME	DESCRIPTION
0x00000004	"dic"	Debug initial command.
0x00000008	"shg"	Stop if the GUI stops responding (that is, hangs).
0x00000010	"htc"	Enable heap tail checking.
0x00000020	"hfc"	Enable heap free checking.
0x00000040	"hpc"	Enable heap parameter checking.
0x00000080	"hvc"	Enable heap validation on call.
0x00000100	"ptc"	Enable pool tail checking.
0x00000200	"pfc"	Enable pool free checking.
0x00000400	"ptg"	Enable pool tagging.
0x00000800	"htg"	Enable heap tagging.
0x00001000	"ust"	Create a user-mode stack trace DB.
0x00002000	"kst"	Create a kernel-mode stack trace DB.
0x00004000	"otl"	Maintain a list of objects for each type.
0x00008000	"htd"	Enable heap tagging by DLL.
0x00010000	"idp"	Unused.
0x00020000	"d32"	Enable debugging of the Microsoft Win32 subsystem.
0x00040000	"ksl"	Enable loading of kernel debugger symbols.

VALUE	NAME	DESCRIPTION
0x00080000	"dps"	Disable paging of kernel stacks.
0x00100000	"scb"	Enable critical system breaks.
0x00200000	"dhc"	Disable heap coalesce on free.
0x00400000	"ece"	Enable close exception.
0x00800000	"eel"	Enable exception logging.
0x01000000	"eot"	Enable object handle type tagging.
0x02000000	"hpa"	Put heap allocations at the end of pages.
0x04000000	"dwl"	Debug WINLOGON.
0x08000000	"ddp"	Disable kernel-mode **DbgPrint** and **KdPrint** output.
0x10000000	NULL	Unused.
0x20000000	"sue"	Stop on unhandled user-mode exception
0x40000000	NULL	Unused.
0x80000000	"dpd"	Disable protected DLL verification.

!gle

6/25/2019 • 2 minutes to read • Edit Online

The **!gle** extension displays the last error value for the current thread.

```
!gle [-all]
```

Parameters

-all

Displays the last error for each user-mode thread on the target system. If you omit this parameter in user mode, the debugger displays the last error for the current thread. If you omit this parameter in kernel mode, the debugger displays the last error for the thread that the current register context specifies.

DLL

Windows 2000	Ext.dll Ntsdexts.dll
Windows XP and later	Ext.dll

Additional Information

For more information about the **GetLastError** routine, see the Micorosft Windows SDK documentation.

Remarks

The **!gle** extension displays the value of **GetLastError** and tries to decode this value.

In kernel mode, the **!gle** extension work only if the debugger can read the thread environment block (TEB).

!gs

12/5/2018 • 2 minutes to read • Edit Online

The **!gs** extension analyzes a /GS stack overflow.

```
!gs
```

DLL

Windows 2000	Ext.dll
Windows XP and later	Ext.dll

Remarks

The **!gs**extension helps debug buffer overruns. Run **!gs**when you encounter a STATUS_STACK_BUFFER_OVERRUN error, as the following example shows.

```
0:000> !gs
Corruption occurred in mshtml!CDoc::OnPaint or one of its callers
Real canary not found at 0x74866010
Canary at gsfailure frame 0x292ea4e7
Corrupted canary 0x0013e2c8: 0x00000000
Corrupted cookie value too generic, skipping init bit-flip check
a caller of mshtml!CDoc::OnPaint has corrupted the EBP from 0x0013e254 to 0x0013
e234
check callers (without canary) of mshtml!CDoc::OnPaint for 0x1 bytes of overflow

The canary doesn't look corrupted. Not sure how we got here
EBP/ESP check skipped: No saved EBP in exception context
Function mshtml!CDoc::OnPaint:
  00000000 - 00000004 this                      CDoc*
 0013de40 - 0013e180 rd                        CDoc::OnPaint::__l39::REGION_DAT
A
  0013e180 - 0013e18c Lock                      CDoc::CLock
  0013e18c - 0013e224 DI                        CFormDrawInfo
  0013e23c - 0013e240 hwndInplace               HWND__*
  0013e240 - 0013e244 prc                       tagRECT*
  0013e248 - 0013e250 ptBefore                  tagPOINT
  0013e250 - 0013e254 fViewIsReady              int
  0013e250 - 0013e254 fHtPalette                int
  0013e254 - 0013e258 fNoPaint                  int
  0013e258 - 0013e260 ptAfter                   tagPOINT
  0013e260 - 0013e264 c                         int
  0013e264 - 0013e268 hrgn                      HRGN__*
  0013e268 - 0013e2a8 ps                        tagPAINTSTRUCT
Candidate buffer : ps 0013e268 to 0013e2a7
  0013e268 ea 04 01 a7 00 00 00 00-10 01 00 00 f3 00 00 00 ................
  0013e278 ed 03 00 00 44 02 00 00-84 e5 13 00 f4 e2 13 00 ....D...........
  ...
  0013e2ac 38 20 01 03 10 e3 13 00-68 6b e6 01 d0 e6 03 00 8 ......hk......
 0013e2bc 80 fa 03 00 0d 00 00 00-10 08 19 00 00 00 00 00 ................
0:000>
```

!handle

8/26/2019 • 4 minutes to read • Edit Online

The **!handle** extension displays information about a handle or handles that one or all processes in the target system own.

User-Mode

```
!handle [Handle [UMFlags [TypeName]]]
!handle -?
```

Kernel-Mode

```
    !handle [Handle [KMFlags [Process [TypeName]]]]
```

Parameters

Handle
Specifies the index of the handle to display. If *Handle* is -1 or if you omit this parameter, the debugger displays data for all handles that are associated with the current process. If *Handle* is 0, the debugger displays data for all handles.

UMFlags
(User mode only) Specifies what the display should contain. This parameter can be a sum of any of the following bit values. (The default value is 0x1.)

Bit 0 (0x1)
Displays handle type information.

Bit 1 (0x2)
Displays basic handle information.

Bit 2 (0x4)
Displays handle name information.

Bit 3 (0x8)
Displays object-specific handle information, when available.

KMFlags
(Kernel mode only) Specifies what the display should contain. This parameter can be a sum of any of the following bit values. (The default value is 0x3.)

Bit 0 (0x1)
Displays basic handle information.

Bit 1 (0x2)
Displays information about objects.

Bit 2 (0x4)
Displays free handle entries. If you do not set this bit and you omit *Handle* or set it to zero, the list of handles that are displayed does not include free handles. If *Handle* specifies a single free handle, it is displayed even if you do not set this bit.

Bit 4 (0x10)
Displays the handle from the kernel handle table instead of the current process.

Bit 5 (0x20)
Interprets the handle as a thread ID or process ID and displays information about the corresponding kernel object.

Process
(Kernel mode only) Specifies a process. You can use the process ID or the hexadecimal address of the process object. This parameter must refer to a currently running process on the target system. If this parameter is -1 or if you omit it, the current process is used. If this parameter is 0, handle information from all processes is displayed.

TypeName
Specifies the type of handle that you want to examine. Only handles that match this type are displayed. *TypeName* is case sensitive. Valid types include Event, Section, File, Port, Directory, SymbolicLink, Mutant, WindowStation, Semaphore, Key, Token, Process, Thread, Desktop, IoCompletion, Timer, Job, and WaitablePort.

-?
(User mode only) Displays some Help text for this extension in the Debugger Command window.

DLL

Windows 2000	Kdextx86.dll Uext.dll Ntsdexts.dll
Windows XP and later	Kdexts.dll Uext.dll Ntsdexts.dll

Additional Information

For more information about handles, see the **!htrace** extension, the Microsoft Windows SDK documentation and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

You can use the **!handle** extension during user-mode and kernel-mode live debugging. You can also use this extension on kernel-mode dump files. However, you cannot use this extension on user-mode dump files, unless you specifically created them with handle information. (You can create such dump files by using the **.dump /mh (Create Dump File)** command.)

During live user-mode debugging, you can use the **.closehandle (Close Handle)** command to close one or more handles.

The following examples are user-mode examples of the **!handle** extension. The following command displays a list of all handles.

```
0:000> !handle
Handle 4
  Type          Section
Handle 8
  Type          Event
Handle c
  Type          Event
Handle 10
  Type          Event
Handle 14
  Type          Directory
Handle 5c
  Type          File
6 Handles
Type            Count
Event           3
Section         1
File            1
Directory       1
```

The following command displays detailed information about handle 0x8.

```
0:000> !handle 8 f
Handle 8
  Type          Event
  Attributes    0
  GrantedAccess 0x100003:
         Synch
         QueryState,ModifyState
  HandleCount   2
  PointerCount  3
  Name          <none>
  Object Specific Information
    Event Type Auto Reset
    Event is Waiting
```

The following examples are kernel-mode examples of **!handle**. The following command lists all handles, including free handles.

```
kd> !handle 0 4
processor number 0
PROCESS 80559800  SessionId: 0  Cid: 0000    Peb: 00000000  ParentCid: 0000
    DirBase: 00039000  ObjectTable: e1000d60  TableSize: 380.
    Image: Idle

New version of handle table at e1002000 with 380 Entries in use

0000: free handle, Entry address e1002000, Next Entry fffffffe
0004: Object: 80ed5238  GrantedAccess: 001f0fff
0008: Object: 80ed46b8  GrantedAccess: 00000000
000c: Object: e1281d00  GrantedAccess: 000f003f
0010: Object: e1013658  GrantedAccess: 00000000
......
0168: Object: ffb6c748  GrantedAccess: 00000003 (Protected)
016c: Object: ff811f90  GrantedAccess: 0012008b
0170: free handle, Entry address e10022e0, Next Entry 00000458
0174: Object: 80dfd5c8  GrantedAccess: 001f01ff
......
```

The following command show detailed information about handle 0x14 in the kernel handle table.

```
kd> !handle 14 13
processor number 0
PROCESS 80559800  SessionId: 0  Cid: 0000    Peb: 00000000  ParentCid: 0000
    DirBase: 00039000  ObjectTable: e1000d60  TableSize: 380.
    Image: Idle

Kernel New version of handle table at e1002000 with 380 Entries in use
0014: Object: e12751d0  GrantedAccess: 0002001f
Object: e12751d0  Type: (80ec8db8) Key
    ObjectHeader: e12751b8
        HandleCount: 1  PointerCount: 1
        Directory Object: 00000000  Name: \REGISTRY\MACHINE\SYSTEM\CONTROLSET001\CONTROL\SESSION
MANAGER\EXECUTIVE
```

The following command shows information about all handles to Section objects in all processes.

```
!handle 0 3 0 Section
...
PROCESS fffffa8004f48940
    SessionId: none  Cid: 0138    Peb: 7f6639bf000  ParentCid: 0004
    DirBase: 10cb74000  ObjectTable: fffff8a00066f700  HandleCount:  39.
    Image: smss.exe

Handle table at fffff8a00066f700 with 39 entries in use

0040: Object: fffff8a000633f00  GrantedAccess: 00000006 (Inherit) Entry: fffff8a000670100
Object: fffff8a000633f00  Type: (fffffa80035fef20) Section
    ObjectHeader: fffff8a000633ed0 (new version)
        HandleCount: 1  PointerCount: 262144
...
```

!heap

10/7/2019 • 15 minutes to read • Edit Online

The **!heap** extension displays heap usage information, controls breakpoints in the heap manager, detects leaked heap blocks, searches for heap blocks, or displays page heap information.

This extension supports the segment heap and the NT heap. Use !heap with no parameter to list all heaps and their type.

```
!heap [HeapOptions] [ValidationOptions] [Heap]
!heap -b [{alloc|realloc|free} [Tag]] [Heap | BreakAddress]
!heap -B {alloc|realloc|free} [Heap | BreakAddress]
!heap -l
!heap -s [SummaryOptions] [StatHeapAddress]
!heap -i HeapAddress
!heap -x [-v] Address
!heap -p [PageHeapOptions]
!heap -srch [Size] Pattern
!heap -flt FilterOptions
!heap -stat [-h Handle [-grp GroupBy [MaxDisplay]]]
!heap [-p] -?
!heap -triage [Handle | Address]
```

Segment and NT Heap Parameters

These parameters work with Segment and NT heaps.

-s
Specifies that summary information is being requested. If *SummaryOptions* and *StatHeapAddress* are omitted, then summary information is displayed for all heaps associated with the current process.

SummaryOptions
Can be any combination of the following options. The *SummaryOptions* are not case-sensitive. Type !heap -s -? for additional information.

OPTION	EFFECT
-v	Verifies all data blocks.
-b *BucketSize*	Specifies the bucket size. The default is 1024 bits.
-d *DumpBlockSize*	Specifies the bucket size.
-a	Dumps all heap blocks.
-c	Specifies that the contents of each block should be displayed.

-triage [*Handle* **|** *Address***]**

Causes the debugger to automatically search for failures in a process's heaps. If a heap handle is specified as an argument, that heap is examined; otherwise, all the heaps are searched for one that contains the given address, and if one is found, it is examined. Using **-triage** is the only way to validate low-fragmentation heap (LFH) corruption.

-x [**-v**]
Causes the debugger to search for the heap block containing the specified address. If -v is added, the command will search the entire virtual memory space of the current process for pointers to this heap block.

-l
Causes the debugger to detect leaked heap blocks.

-i *Address* **-h** *HeapAddress*
Displays information about the specified *Heap*.

Address
Specifies the address to search for.

-?
Displays some brief Help text for this extension in the Debugger Command window. Use **!heap -?** for generic help, and **!heap -p -?** for page heap help.

NT Heap Parameters

These parameters work only with the NT Heap.

HeapOptions
Can be any combination of the following options. The *HeapOptions* values are case-sensitive.

OPTION	EFFECT
-v	Causes the debugger to validate the specified heap. **Note** This option does not detect low fragmentation heap (LFH) corruption. Use **-triage** instead.
-a	Causes the display to include all information for the specified heap. Size, in this case, is rounded up to the heap granularity. (Running **!heap** with the **-a** option is equivalent to running it with the three options **-h -f -m**, which can take a long time.)
-h	Causes the display to include all non-LFH entries for the specified heap.
-hl	Causes the display to include all the entries for the specified heap(s), including LFH entries.
-f	Causes the display to include all the free list entries for the specified heap.

OPTION	EFFECT
-m	Causes the display to include all the segment entries for the specified heap.
-t	Causes the display to include the tag information for the specified heap.
-T	Causes the display to include the pseudo-tag entries for the specified heap.
-g	Causes the display to include the global tag information. Global tags are associated with each untagged allocation.
-s	Causes the display to include summary information for the specified heap.
-k	(x86-based targets only) Causes the display to include the stack backtrace associated with each entry.

ValidationOptions
Can be any single one of the following options. The *ValidationOptions* are case-sensitive.

OPTION	EFFECT
-D	Disables validate-on-call for the specified heap.
-E	Enables validate-on-call for the specified heap.
-d	Disables heap checking for the specified heap.
-e	Enables heap checking for the specified heap.

-i *Heap Address* **or** *HeapAddress*
Displays information about the specified *Heap*.

BreakAddress
Specifies the address of a block where a breakpoint is to be set or removed.

-b
Causes the debugger to create a conditional breakpoint in the heap manager. The **-b** option can be followed by **alloc**, **realloc**, or **free**; this specifies whether the breakpoint will be activated by allocating, reallocating, or freeing memory. If *BreakAddress* is used to specify the address of the block, the breakpoint type can be omitted. If *Heap* is used to specify the heap address or heap index, the type must be included, as well as the *Tag* parameter.

Tag
Specifies the tag name within the heap.

-B

Causes the debugger to remove a conditional breakpoint from the heap manager. The breakpoint type (**alloc**, **realloc**, or **free**) must be specified, and must be the same as that used with the **-b** option.

StatHeapAddress
Specifies the address of the heap. If this is 0 or omitted, all heaps associated with the current process are displayed.

-p
Specifies that page heap information is being requested. If this is used without any *PageHeapOptions*, all page heaps will be displayed.

PageHeapOptions
Can be any single one of the following options. The *PageHeapOptions* are case-sensitive. If no options are specified, then all possible page heap handles will be displayed.

OPTION	EFFECT
-h *Handle*	Causes the debugger to display detailed information about a page heap with handle *Handle*.
-a *Address*	Causes the debugger to find the page heap whose block contains *Address*. Full details of how this address relates to full-page heap blocks will be included, such as whether this address is part of a page heap, its offset inside the block, and whether the block is allocated or was freed. Stack traces are included whenever available. When using this option, size is displayed in multiples of the heap allocation granularity.
-t[**c**|**s**] [*Traces*]	Causes the debugger to display the collected traces of the heavy heap users. *Traces* specifies the number of traces to display; the default is four. If there are more traces than the specified number, the earliest traces are displayed. If **-t** or **-tc** is used, the traces are sorted by count usage. If **-ts** is used, the traces are sorted by size. (The **-tc** and **-ts** options are supported in Windows XP only; the **-t** option is supported only in Windows XP and earlier versions of Windows.)
-fi [*Traces*]	Causes the debugger to display the most recent fault injection traces. *Traces* specifies the quantity to be displayed; the default is 4.
-all	Causes the debugger to display detailed information about all page heaps.
-?	Causes the debugger to display page heap help, including a diagram of heap blocks. (These diagrams can also be seen in the following Remarks section.)

Before you can use any **!heap -p** extension command, the page heap must be enabled for your target process. See details in the following Remarks section.

-srch
Scans all heaps for the given pattern.

Pattern
Specifies a pattern for which to look.

Size
Can be any single one of the following options. This specifies the size of the pattern. The '-' is required.

OPTION	EFFECT
-b	The pattern is one BYTE in size.
-w	The pattern is one WORD in size.
-d	The pattern is one DWORD in size.
-q	The pattern is one QWORD in size.

If none of the above are specified, then the pattern is assumed to be of the same size as the machine pointer.

-flt
Limits the display to include only heaps with the specified size or size range.

FilterOptions
Can be any single one of the following options. The *FilterOptions* are case-sensitive.

OPTION	EFFECT
s *Size*	Limits the display to include only heaps of the specified size.
r *SizeMin SizeMax*	Limits the display to include only heaps within the specified size range.

-stat
Displays usage statistics for the specified heap.

-h *Handle*
Causes usage statistics for only the heap at *Handle* to be displayed. If *Handle* is 0 or omitted, then usage statistics for all heaps are displayed.

-grp *GroupBy*
Reorders the display as specified by *GroupBy*. The options for *GroupBy* can be found in the following table.

OPTION	EFFECT
A	Displays the usage statistics according to allocation size.
B	Displays the usage statistics according to block count.

OPTION	EFFECT
S	Displays the usage statistics according to the total size of each allocation.

MaxDisplay
Limits the output to only *MaxDisplay* number of lines.

DLL

Windows XP and later	Ext.dll Exts.dll

Additional Information

For information about heaps, see the following resources:

Book: *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Example 11: Enabling Page Heap Verification

Example 12: Using Page Heap Verification to Find a Bug

For information on using the Heap memory process logger, see Example 11: Starting a Private Trace Session

Remarks

This extension command can be used to perform a variety of tasks.

The standard **!heap** command is used to display heap information for the current process. (This should be used only for user-mode processes. The **!pool** extension command should be used for system processes.)

The **!heap -b** and **!heap -B** commands are used to create and delete conditional breakpoints in the heap manager.

The **!heap -l** command detects leaked heap blocks. It uses a garbage collector algorithm to detect all busy blocks from the heaps that are not referenced anywhere in the process address space. For huge applications, it can take a few minutes to complete. This command is only available in Windows XP and later versions of Windows.

The **!heap -x** command searches for a heap block containing a given address. If the **-v** option is used, this command will additionally search the entire virtual memory space of the current process for pointers to this heap block. This command is only available in Windows XP and later versions of Windows.

The **!heap -p** command displays various forms of page heap information. Before using **!heap -p**, you must enable the page heap for the target process. This is done through the Global Flags (gflags.exe) utility. To do this, start the utility, fill in the name of the target application in the **Image File Name** text box, select **Image File Options** and **Enable page heap**, and click **Apply**. Alternatively, you can start the Global Flags utility from a Command Prompt window by typing **gflags /i** *xxx.exe* **+hpa**, where *xxx.exe* is the name of the target application.

The **!heap -p -t[c|s]** commands are not supported beyond Windows XP. Use the UMDH tool provided with the debugger package to obtain similar results.

The **!heap -srch** command displays those heap entries that contain a certain specified pattern.

The **!heap -flt** command limits the display to only heap allocations of a specified size.

The **!heap -stat** command displays heap usage statistics.

Here is an example of the standard **!heap** command:

```
0:000> !ntsdexts.heap -a
```

```
0:000> !ntsdexts.heap -a
Index   Address  Name      Debugging options enabled
  1:   00250000
    Segment at 00250000 to 00350000 (00056000 bytes committed)
    Flags:               50000062
    ForceFlags:          40000060
    Granularity:         8 bytes
    Segment Reserve:     00100000
    Segment Commit:      00004000
    DeCommit Block Thres:00000400
    DeCommit Total Thres:00002000
    Total Free Size:     000003be
    Max. Allocation Size:7ffddfff
    Lock Variable at:    00250b54
    Next TagIndex:       0012
    Maximum TagIndex:    07ff
    Tag Entries:         00350000
    PsuedoTag Entries:   00250548
    Virtual Alloc List:  00250050
    UCR FreeList:        002504d8
    128-bit bitmap of free lists
    FreeList Usage:      00000014 00000000 00000000 00000000
              Free    Free
              List    List
#       Head      Blink      Flink
    FreeList[ 00 ] at 002500b8: 002a4378 . 002a4378
                                0x02 - HEAP_ENTRY_EXTRA_PRESENT
                                0x04 - HEAP_ENTRY_FILL_PATTERN
        Entry     Prev    Cur   0x10 - HEAP_ENTRY_LAST_ENTRY

Address   Size    Size  flags
002a4370: 00098 . 01c90 [14] - free
    FreeList[ 02 ] at 002500c8: 0025cb30 . 002527b8
002527b0: 00058 . 00010 [04] - free
0025cb28: 00088 . 00010 [04] - free
    FreeList[ 04 ] at 002500d8: 00269a08 . 0026e530
0026e528: 00038 . 00020 [04] - free
0026a4d0: 00038 . 00020 [06] - free
0026f9b8: 00038 . 00020 [04] - free
0025cda0: 00030 . 00020 [06] - free
00272660: 00038 . 00020 [04] - free
0026ab60: 00038 . 00020 [06] - free
00269f20: 00038 . 00020 [06] - free
00299818: 00038 . 00020 [04] - free
0026c028: 00038 . 00020 [06] - free
00269a00: 00038 . 00020 [46] - free

    Segment00 at 00250b90:
Flags:           00000000
Base:            00250000
First Entry:     00250bc8
Last Entry:      00350000
Total Pages:     00000080
Total UnCommit:  00000055
Largest UnCommit:000aa000
UnCommitted Ranges: (1)
    002a6000: 000aa000

    Heap entries for Segment00 in Heap 250000
                         0x01 - HEAP_ENTRY_BUSY
                         0x02 - HEAP_ENTRY_EXTRA_PRESENT
                         0x04 - HEAP_ENTRY_FILL_PATTERN
                         0x08 - HEAP_ENTRY_VIRTUAL_ALLOC
                         0x10 - HEAP_ENTRY_LAST_ENTRY
                         0x20 - HEAP_ENTRY_SETTABLE_FLAG1
                         0x40 - HEAP_ENTRY_SETTABLE_FLAG2
Entry     Prev    Cur   0x80 - HEAP_ENTRY_SETTABLE_FLAG3

Address   Size    Size  flags       (Bytes used)    (Tag name)
```

```
00250000: 00000 . 00b90 [01] - busy (b90)
00250b90: 00b90 . 00038 [01] - busy (38)
00250bc8: 00038 . 00040 [07] - busy (24), tail fill (NTDLL!LDR Database)
00250c08: 00040 . 00060 [07] - busy (48), tail fill (NTDLL!LDR Database)
00250c68: 00060 . 00028 [07] - busy (10), tail fill (NTDLL!LDR Database)
00250c90: 00028 . 00060 [07] - busy (48), tail fill (NTDLL!LDR Database)
00250cf0: 00060 . 00050 [07] - busy (38), tail fill (Objects=  80)
00250d40: 00050 . 00048 [07] - busy (2e), tail fill (NTDLL!LDR Database)
00250d88: 00048 . 00c10 [07] - busy (bf4), tail fill (Objects>1024)
00251998: 00c10 . 00030 [07] - busy (12), tail fill (NTDLL!LDR Database)
...
002525c0: 00030 . 00060 [07] - busy (48), tail fill (NTDLL!LDR Database)
00252620: 00060 . 00050 [07] - busy (38), tail fill (NTDLL!LDR Database)
00252670: 00050 . 00040 [07] - busy (22), tail fill (NTDLL!CSRSS Client)
002526b0: 00040 . 00040 [07] - busy (24), tail fill (Objects=  64)
002526f0: 00040 . 00040 [07] - busy (24), tail fill (Objects=  64)
00252730: 00040 . 00028 [07] - busy (10), tail fill (Objects=  40)
00252758: 00028 . 00058 [07] - busy (3c), tail fill (Objects=  88)
002527b0: 00058 . 00010 [04] free fill
002527c0: 00010 . 00058 [07] - busy (3c), tail fill (NTDLL!LDR Database)
00252818: 00058 . 002d0 [07] - busy (2b8), tail fill (Objects= 720)
00252ae8: 002d0 . 00330 [07] - busy (314), tail fill (Objects= 816)
00252e18: 00330 . 00330 [07] - busy (314), tail fill (Objects= 816)
00253148: 00330 . 002a8 [07] - busy (28c), tail fill (NTDLL!LocalAtom)
002533f0: 002a8 . 00030 [07] - busy (18), tail fill (NTDLL!LocalAtom)
00253420: 00030 . 00030 [07] - busy (18), tail fill (NTDLL!LocalAtom)
00253450: 00030 . 00098 [07] - busy (7c), tail fill (BASEDLL!LMEM)
002534e8: 00098 . 00060 [07] - busy (44), tail fill (BASEDLL!TMP)
00253548: 00060 . 00020 [07] - busy (1), tail fill (Objects=  32)
00253568: 00020 . 00028 [07] - busy (10), tail fill (Objects=  40)
00253590: 00028 . 00030 [07] - busy (16), tail fill (Objects=  48)
...
0025ccb8: 00038 . 00060 [07] - busy (48), tail fill (NTDLL!LDR Database)
0025cd18: 00060 . 00058 [07] - busy (3c), tail fill (NTDLL!LDR Database)
0025cd70: 00058 . 00030 [07] - busy (18), tail fill (NTDLL!LDR Database)
0025cda0: 00030 . 00020 [06] free fill (NTDLL!Temporary)
0025cdc0: 00020 . 00258 [07] - busy (23c), tail fill (Objects= 600)
0025d018: 00258 . 01018 [07] - busy (1000), tail fill (Objects>1024)
0025e030: 01018 . 00060 [07] - busy (48), tail fill (NTDLL!LDR Database)
...
002a4190: 00028 . 00118 [07] - busy (100), tail fill (BASEDLL!GMEM)
002a42a8: 00118 . 00030 [07] - busy (18), tail fill (Objects=  48)
002a42d8: 00030 . 00098 [07] - busy (7c), tail fill (Objects= 152)
002a4370: 00098 . 01c90 [14] free fill
002a6000:      000aa000      - uncommitted bytes.
```

Here is an example of the **!heap -l** command:

```
1:0:011> !heap -l
1:Heap 00170000
Heap 00280000
Heap 00520000
Heap 00b50000
Heap 00c60000
Heap 01420000
Heap 01550000
Heap 016d0000
Heap 019b0000
Heap 01b40000
Scanning VM ...
## Entry     User      Heap      Segment       Size  PrevSize  Flags

001b2958  001b2960  00170000  00000000        40        18  busy extra
001b9cb0  001b9cb8  00170000  00000000        80       300  busy extra
001ba208  001ba210  00170000  00000000        80        78  busy extra
001cbc90  001cbc98  00170000  00000000        e0        48  busy extra
001cbd70  001cbd78  00170000  00000000        d8        e0  busy extra
001cbe90  001cbe98  00170000  00000000        68        48  busy extra
001cbef8  001cbf00  00170000  00000000        58        68  busy extra
001cc078  001cc080  00170000  00000000        f8       128  busy extra
001cc360  001cc368  00170000  00000000        80        50  busy extra
001cc3e0  001cc3e8  00170000  00000000        58        80  busy extra
001fe550  001fe558  00170000  00000000       150       278  busy extra
001fe6e8  001fe6f0  00170000  00000000        48        48  busy extra
002057a8  002057b0  00170000  00000000        58        58  busy extra
00205800  00205808  00170000  00000000        48        58  busy extra
002058b8  002058c0  00170000  00000000        58        70  busy extra
00205910  00205918  00170000  00000000        48        58  busy extra
00205958  00205960  00170000  00000000        90        48  busy extra
00246970  00246978  00170000  00000000        60        88  busy extra
00251168  00251170  00170000  00000000        78        d0  busy extra user_flag
00527730  00527738  00520000  00000000        40        40  busy extra
00527920  00527928  00520000  00000000        40        80  busy extra
21 leaks detected.
```

The table in this example contains all 21 leaks found.

Here is an example of the **!heap -x** command:

```
0:011> !heap 002057b8 -x
## Entry     User      Heap      Segment       Size  PrevSize  Flags

002057a8  002057b0  00170000  00170640        58        58  busy extra
```

Here is an example of the **!heap -x -v** command:

```
1:0:011> !heap 002057b8 -x -v
## 1:Entry     User      Heap      Segment       Size  PrevSize  Flags

002057a8  002057b0  00170000  00170640        58        58  busy extra

Search VM for address range 002057a8 - 002057ff : 00205990 (002057d0),
```

In this example, there is a pointer to this heap block at address 0x00205990.

Here is an example of the **!heap -flt s** command:

```
0:001>!heap -flt s 0x50
```

This will display all of the allocations of size 0x50.

Here is an example of the **!heap -flt r** command:

```
0:001>!heap -flt r 0x50 0x80
```

This will display each allocation whose size is between 0x50 and 0x7F.

Here is an example of the **!heap -srch** command.

```
0:001> !heap -srch 77176934
    _HEAP @ 00090000
   in HEAP_ENTRY: Size : Prev Flags - UserPtr UserSize - state
        00099A48: 0018 : 0005 [01] - 00099A50 (000000B8) - (busy)
          ole32!CALLFRAME_CACHE<INTERFACE_HELPER_CLSID>::`vftable'
    _HEAP @ 00090000
   in HEAP_ENTRY: Size : Prev Flags - UserPtr UserSize - state
        00099B58: 0018 : 0005 [01] - 00099B60 (000000B8) - (busy)
          ole32!CALLFRAME_CACHE<INTERFACE_HELPER_CLSID>::`vftable'
```

The following diagrams show the arrangement of heap blocks.

Light page heap block -- allocated:

```
 +-----+---------------+---+
 |     |               |   |
 +-----+---------------+---+
    ^         ^          ^
    |         |          8 suffix bytes (filled with 0xA0)
    |         User allocation (filled with E0 if zeroing not requested)
    Block header (starts with 0xABCDAAAA and ends with 0xDCBAAAAA)
```

Light page heap block -- freed:

```
 +-----+---------------+---+
 |     |               |   |
 +-----+---------------+---+
    ^         ^          ^
    |         |          8 suffix bytes (filled with 0xA0)
    |         User allocation (filled with F0 bytes)
    Block header (starts with 0xABCDAAA9 and ends with 0xDCBAAA9)
```

Full page heap block -- allocated:

```
 +-----+---------+---+-------
 |     |         |   |  ... N/A page
 +-----+---------+---+-------
    ^       ^      ^
    |       |      0-7 suffix bytes (filled with 0xD0)
    |       User allocation (if zeroing not requested, filled
            with C0)
    Block header (starts with 0xABCDBBBB and ends with 0xDCBABBBB)
```

Full page heap block -- freed:

```
+-----+---------+---+-------
|     |         |   |  ... N/A page
+-----+---------+---+-------
   ^       ^      ^
   |       |      0-7 suffix bytes (filled with 0xD0)
   |       User allocation (filled with F0 bytes)
   Block header (starts with 0xABCDBBA and ends with 0xDCBABBBA)
```

To see the stack trace of the allocation or the freeing of a heap block or full page heap block, use **dt DPH_BLOCK_INFORMATION** with the header address, followed by **dds** with the block's **StackTrace** field.

!help

12/5/2018 • 2 minutes to read • Edit Online

The **!help** extension displays help text that describes the extension commands exported from the extension DLL.

Do not confuse this extension command with the **? (Command Help)** or **.help (Meta-Command Help)** commands.

```
![ExtensionDLL.]help [-v] [CommandName]
```

Parameters

ExtensionDLL
Displays help for the specified extension DLL. Type the name of an extension DLL without the .dll file name extension. If the DLL file is not in the extension search path (as displayed by using **.chain (List Debugger Extensions)**), include the path to the DLL file. For example, to display help for uext.dll, type **!uext.help** or **!***Path***\winext\uext.help**.

If you omit the *ExtensionDLL*, the debugger will display the help text for the first extension DLL in the list of loaded DLLs.

-v
Displays the most detailed help text available. This feature is not supported in all DLLs.

CommandName
Displays only the help text for the specified command. This feature is not supported in all DLLs or for all commands.

DLL

This extension is supported by most extension DLLs.

Remarks

Some individual commands will also display a help text if you use the **/?** or **-?** parameter with the command name.

!homedir

12/5/2018 • 2 minutes to read • Edit Online

The **!homedir** extension sets the default directory used by the symbol server and the source server.

```
!homedir Directory
!homedir
```

Parameters

Directory
Specifies the new directory to use as the home directory.

DLL

Windows 2000	Dbghelp.dll
Windows XP and later	Dbghelp.dll

Remarks

If the **!homedir** extension is used with no argument, the current home directory is displayed.

The cache for a source server is located in the src subdirectory of the home directory. The downstream store for a symbol server defaults to the sym subdirectory of the home directory, unless a different location is specified.

When WinDbg is started, the home directory is the directory where Debugging Tools for Windows was installed. The **!homedir** extension can be used to change this value.

!hstring

12/5/2018 • 2 minutes to read • Edit Online

The **!hstring** extension displays the fields of an **HSTRING**. The last item in the display is the string itself.

```
!hstring Address
```

Parameters

Address
The address of an **HSTRING**.

Remarks

The **HSTRING** data type supports strings that have embedded NULL characters. However, the **!hstring** extension displays the string only up to the first NULL character. To see the entire string including the embedded NULL characters, use the **!hstring2** extension.

See also

Windows Runtime Debugger Commands

!hstring2

!winrterr

!hstring2

12/5/2018 • 2 minutes to read • Edit Online

The **!hstring2** extension displays an entire **HSTRING** including any embedded NULL characters in the string itself.

```
!hstring2 Address
```

Parameters

Address
The address of an **HSTRING**.

See also

Windows Runtime Debugger Commands

!hstring

!winrterr

!htrace

8/26/2019 • 2 minutes to read • Edit Online

The **!htrace** extension displays stack trace information for one or more handles.

User-Mode Syntax

```
!htrace [Handle [Max_Traces]]
!htrace -enable [Max_Traces]
!htrace -snapshot
!htrace -diff
!htrace -disable
!htrace -?
```

Kernel-Mode Syntax

```
    !htrace [Handle [Process [Max_Traces]]]
!htrace -?
```

Parameters

Handle
Specifies the handle whose stack trace will be displayed. If *Handle* is 0 or omitted, stack traces for all handles in the process will be displayed.

Process
(Kernel mode only) Specifies the process whose handles will be displayed. If *Process* is 0 or omitted, then the current process is used. In user mode, the current process is always used.

Max_Traces
Specifies the maximum number of stack traces to display. In user mode, if this parameter is omitted, then all the stack traces for the target process will be displayed.

-enable
(User mode only) Enables handle tracing and takes the first snapshot of the handle information to use as the initial state by the **-diff** option.

-snapshot
(User mode only) Takes a snapshot of the current handle information to use as the initial state by the **-diff** option.

-diff
(User mode only) Compares current handle information with the last snapshot of handle information that was taken. Displays all handles that are still open.

-disable
(User mode only; Windows Server 2003 and later only) Disables handle tracing. In Windows XP, handle tracing can be disabled only by terminating the target process.

-?
Displays some brief Help text for this extension in the Debugger Command window.

DLL

Windows 2000	Unavailable
Windows XP and later	Kdexts.dll Ntsdexts.dll

Additional Information

For information about handles, see the Microsoft Windows SDK documentation and *Microsoft Windows Internals* by Mark Russinovich and David Solomon. To display further information about a specific handle, use the **!handle** extension.

Remarks

Before **!htrace** can be used, handle tracing must be enabled. One way to enable handle tracing is to enter the **!htrace -enable** command. When handle tracing is enabled, stack trace information is saved each time the process opens a handle, closes a handle, or references an invalid handle. It is this stack trace information that **!htrace** displays.

Note You can also enable handle tracing by activating Application Verifier for the target process and selecting the **Handles** option.

Some of the traces reported by **!htrace** may be from a different process context. In this case, the return addresses may not resolve properly in the current process context, or may resolve to the wrong symbols.

The following example displays information about all handles in process 0x81400300:

```
kd> !htrace 0 81400300
Process 0x81400300
ObjectTable 0xE10CCF60
##

Handle 0x7CC - CLOSE:
0x8018FCB9: ntoskrnl!ExDestroyHandle+0x103
0x801E1D12: ntoskrnl!ObpCloseHandleTableEntry+0xE4
0x801E1DD9: ntoskrnl!ObpCloseHandle+0x85
0x801E1EDD: ntoskrnl!NtClose+0x19
0x010012C1: badhandle!mainCRTStartup+0xE3
## 0x77DE0B2F: KERNEL32!BaseProcessStart+0x3D

Handle 0x7CC - OPEN:
0x8018F44A: ntoskrnl!ExCreateHandle+0x94
0x801E3390: ntoskrnl!ObpCreateUnnamedHandle+0x10C
0x801E7317: ntoskrnl!ObInsertObject+0xC3
0x77DE23B2: KERNEL32!CreateSemaphoreA+0x66
0x010011C5: badhandle!main+0x45
0x010012C1: badhandle!mainCRTStartup+0xE3
## 0x77DE0B2F: KERNEL32!BaseProcessStart+0x3D

Handle 0x7DC - BAD REFERENCE:
0x8018F709: ntoskrnl!ExMapHandleToPointerEx+0xEA
0x801E10F2: ntoskrnl!ObReferenceObjectByHandle+0x12C
0x801902BE: ntoskrnl!NtSetEvent+0x6C
0x80154965: ntoskrnl!_KiSystemService+0xC4
0x010012C1: badhandle!mainCRTStartup+0xE3
## 0x77DE0B2F: KERNEL32!BaseProcessStart+0x3D

Handle 0x7DC - CLOSE:
0x8018FCB9: ntoskrnl!ExDestroyHandle+0x103
0x801E1D12: ntoskrnl!ObpCloseHandleTableEntry+0xE4
0x801E1DD9: ntoskrnl!ObpCloseHandle+0x85
0x801E1EDD: ntoskrnl!NtClose+0x19
0x010012C1: badhandle!mainCRTStartup+0xE3
## 0x77DE0B2F: KERNEL32!BaseProcessStart+0x3D

Handle 0x7DC - OPEN:
0x8018F44A: ntoskrnl!ExCreateHandle+0x94
0x801E3390: ntoskrnl!ObpCreateUnnamedHandle+0x10C
0x801E7317: ntoskrnl!ObInsertObject+0xC3
0x77DE265C: KERNEL32!CreateEventA+0x66
0x010011A0: badhandle!main+0x20
0x010012C1: badhandle!mainCRTStartup+0xE3
0x77DE0B2F: KERNEL32!BaseProcessStart+0x3D
##

Parsed 0x6 stack traces.
Dumped 0x5 stack traces.
```

!imggp

12/5/2018 • 2 minutes to read • Edit Online

The **!imggp** extension displays the global pointer (GP) directory entry value for a 64-bit image.

```
!imggp Address
```

Parameters

Address
Specifies the base address of the image.

DLL

Windows 2000	Ext.dll
Windows XP and later	Ext.dll

!imgreloc

12/5/2018 • 2 minutes to read • Edit Online

The **!imgreloc** extension displays the addresses of each loaded module and indicates their former addresses before they were relocated.

```
!imgreloc Address
```

Parameters

Address
Specifies the base address of the image.

DLL

Windows 2000	Ext.dll
Windows XP and later	Ext.dll

Remarks

Here is an example:

```
0:000> !imgreloc 00400000
00400000 Prymes - at preferred address
010e0000 appvcore - RELOCATED from 00400000
5b2f0000 verifier - at preferred address
5d160000 ShimEng - at preferred address
```

!kuser

12/5/2018 • 2 minutes to read • Edit Online

The **!kuser** extension displays the shared user-mode page (KUSER_SHARED_DATA).

```
!kuser
```

DLL

Windows 2000	Kdextx86.dll Ntsdexts.dll
Windows XP and later	Exts.dll

Remarks

The KUSER_SHARED_DATA page gives resource and other information about the user who is currently logged on.

Here is an example. Note that, in this example, the tick count is displayed in both its raw form and in a more user-friendly form, which is in parentheses. The user-friendly display is available only in Windows XP and later.

```
kd> !kuser
_KUSER_SHARED_DATA at 7ffe0000
TickCount:    fa00000 * 00482006 (0:20:30:56.093)
TimeZone Id: 2
ImageNumber Range: [14c .. 14c]
Crypto Exponent: 0
SystemRoot: 'F:\WINDOWS'
```

!list

12/5/2018 • 3 minutes to read • Edit Online

The **!list** extension executes the specified debugger commands repeatedly, once for every element in a linked list.

```
!list -t [Module!]Type.Field -x "Commands" [-a "Arguments"] [Options] StartAddress
!list " -t [Module!]Type.Field -x \"Commands\" [-a \"Arguments\"] [Options] StartAddress "
!list -h
```

Parameters

Module
An optional parameter specifying the module that defines this structure. If there is a chance that *Type* may match a valid symbol in a different module, you should include *Module* to eliminate the ambiguity.

Type
Specifies the name of a data structure.

Field
Specifies the field containing the list link. This can actually be a sequence of fields separated by periods (in other words, **Type.Field.Subfield.Subsubfield**, and so on).

-x "*Commands***"**
Specifies the commands to execute. This can be any combination of debugger commands. It must be enclosed in quotation marks. If multiple commands are specified, separate them with semicolons, enclose the entire collection of **!list** arguments in quotation marks, and use an escape character (\ ) before each quotation mark inside these outer quotation marks. If *Commands* is omitted, the default is **dp (Display Memory)**.

-a "*Arguments***"**
Specifies the arguments to pass to the *Commands* parameter. This must be enclosed in quotation marks. *Arguments* can be any valid argument string that would normally be allowed to follow this command, except that *Arguments* cannot contain quotation marks. If the pseudo-register **$extret** is included in *Commands*, the **-a "***Arguments***"** parameter can be omitted.

Options
Can be any number of the following options:

-e
Echoes the command being executed for each element.

-m *Max*
Specifies the maximum number of elements to execute the command for.

StartAddress
Specifies the address of the first data structure. This is the address at the top of the structure, not necessarily the address of the link field.

-h
Displays some brief Help text for this extension in the Debugger Command window.

DLL

Windows 2000	Ext.dll
Windows XP and later	Ext.dll

Remarks

The **!list** extension will go through the linked list and issue the specified command once for each list element.

The pseudo-register **$extret** is set to the value of the list-entry address for each list element. For each element, the command string *Commands* is executed. This command string can reference this pseudo-register using the **$extret** syntax. If this does not appear in the command string, the value of the list-entry address is appended to the end of the command string before execution. If you need to specify where this value should appear in your command, you must specify this pseudo-register explicitly.

This command sequence will run until the list terminates in a null pointer, or terminates by looping back onto the first element. If the list loops back onto a later element, this command will not stop. However, you can stop this command at any time by using **CTRL+C** in KD and CDB, or Debug | Break or CTRL+BREAK in WinDbg.

Each time a command is executed, the address of the current structure will be used as the *default address* if the command being used has optional address parameters.

Following are two examples of how to use this command in user mode. Note that kernel mode usage is also possible but follows a different syntax.

As a simple example, assume that you have a structure whose type name is **MYTYPE**, and which has links within its **.links.Flink** and **.links.Blink** fields. You have a linked list that begins with the structure at 0x6BC000. The following extension command will go through the list and for each element will execute a **dd** L2 command. Because no address is being specified to the **dd** command, it will take the address of the list head as the desired address. This causes the first two DWORDs in each structure to be displayed.

```
0:000> !list -t MYTYPE.links.Flink -x "dd" -a "L2" 0x6bc00
```

As a more complex example, consider the case of using **$extret**. It follows the list of type _LIST_ENTRY at **RtlCriticalSectionList**. For each element, it displays the first four DWORDS, and then displays the _RTL_CRITICAL_SECTION_DEBUG structure located at an offset of eight bytes prior to the **Flink** element of the list entry.

```
0:000> !list "-t ntdll!_LIST_ENTRY.Flink -e -x \"dd @$extret l4; dt ntdll!_RTL_CRITICAL_SECTION_DEBUG
@$extret-0x8\" ntdll!RtlCriticalSectionList"
dd @$extret l4; dt ntdll!_RTL_CRITICAL_SECTION_DEBUG @$extret-0x8
7c97c0c8  7c97c428 7c97c868 01010000 00000080
   +0x000 Type             : 1
   +0x002 CreatorBackTraceIndex : 0
   +0x004 CriticalSection  : (null)
   +0x008 ProcessLocksList : _LIST_ENTRY [ 0x7c97c428 - 0x7c97c868 ]
   +0x010 EntryCount       : 0x1010000
   +0x014 ContentionCount  : 0x80
   +0x018 Spare            : [2] 0x7c97c100

dd @$extret l4; dt ntdll!_RTL_CRITICAL_SECTION_DEBUG @$extret-0x8
7c97c428  7c97c448 7c97c0c8 00000000 00000000
   +0x000 Type             : 0
   +0x002 CreatorBackTraceIndex : 0
   +0x004 CriticalSection  : 0x7c97c0a0
   +0x008 ProcessLocksList : _LIST_ENTRY [ 0x7c97c448 - 0x7c97c0c8 ]
   +0x010 EntryCount       : 0
   +0x014 ContentionCount  : 0
   +0x018 Spare            : [2] 0
```

!lmi

12/5/2018 • 2 minutes to read • Edit Online

The **!lmi** extension displays detailed information about a module.

```
!lmi Module
```

Parameters

Module
Specifies a loaded module, either by name or by base address.

DLL

Windows 2000	Dbghelp.dll
Windows XP and later	Dbghelp.dll

Remarks

Module addresses can be determined by using the **lm (List Loaded Modules)** command.

The **!lmi** extension analyzes the module headers and displays a formatted summary of the information therein. If the module headers are paged out, an error message is displayed. To see a more extensive display of header information, use the **!dh** extension command.

This command shows a number of fields, each with a different title. Some of these titles have specific meanings:

- The **Image Name** field shows the name of the executable file, including the extension. Typically, the full path is included in user mode but not in kernel mode.
- The **Module** field shows the *module name*. This is usually just the file name without the extension. In a few cases, the module name differs significantly from the file name.
- The **Symbol Type** field shows information about the debugger's attempts to load this module's symbols. For an explanation of the various status values, see Symbol Status Abbreviations. If symbols have been loaded, the symbol file name follows this.
- The first address in the module is shown as **Base Address**. The size of the module is shown as **Size**. Thus, if **Base Address** is "faab4000" and **Size** is "2000", the module extends from 0xFAAB4000 to 0xFAAB5FFF, inclusive.

Here is an example:

```
0:000> lm
start    end        module name
00400000 0042d000   Prymes     C (pdb symbols)              Prymes.pdb
77e80000 77f35000   KERNEL32     (export symbols)           C:\WINNT\system32\KERNEL32.dll
77f80000 77ffb000   ntdll        (export symbols)           ntdll.dll

0:000> !lmi 00400000
Loaded Module Info: [00400000]
         Module: Prymes
   Base Address: 00400000
     Image Name: Prymes.exe
   Machine Type: 332 (I386)
     Time Stamp: 3c76c346 Fri Feb 22 14:16:38 2002
           Size: 2d000
       CheckSum: 0
Characteristics: 230e stripped
Debug Data Dirs: Type Size     VA  Pointer
                 MISC  110,     0,   77a00 [Data not mapped]
    Symbol Type: EXPORT   - PDB not found
    Load Report: export symbols
```

For an explanation of the abbreviations shown on the **Characteristics** line of this example, see Symbol Status Abbreviations.

!mui

12/5/2018 • 2 minutes to read • Edit Online

The **!mui** extension displays the Multilingual User Interface (MUI) cache information.

```
!mui -c
!mui -s
!mui -r ModuleAddress
!mui -i
!mui -f
!mui -t
!mui -u
!mui -d ModuleAddress
!mui -e ModuleAddress
!mui -?
```

Parameters

-c
Causes the output to include the language identifier (ID), a pointer to the module, a pointer to the resource configuration data, and a pointer to the associated MUI DLL for each module.

-s
(Kernel Mode Only) Causes the display to include the full file paths for the module and associated MUI DLL for each module.

-r **** *ModuleAddress*
Causes the resource configuration data for the module at *ModuleAddress* to be displayed. This includes the file type, the checksum value, and the resource types.

-i
Causes the output to include the installed and licensed MUI languages and their associated information.

-f
Causes the output to include the loader merged language fallback list.

-t
Causes the output to include the thread preference language.

-u
Causes the output to include the current thread user UI language setting.

-d **** *ModuleAddress*
Causes the output to include contained resources for the module at *ModuleAddress*.

-e **** *ModuleAddress*
Causes the output to include contained resource types for the module at *ModuleAddress*.

-?
Displays some brief Help text for this extension in the Debugger Command window.

DLL

Windows XP	Unavailable
Windows Vista and later	Exts.dll

Additional Information

For information about MUI and resource configuration data format, see the Microsoft Windows SDK documentation.

!net_send

12/5/2018 • 2 minutes to read • Edit Online

The **!net_send** extension sends a message over a local network.

```
!net_send SendingMachine TargetMachine Sender Message
```

Parameters

SendingMachine
Specifies the computer that will process the command. It is recommended that this be the name of the computer that the debugger is running on, since your network configuration may refuse to send the message otherwise. *SendingMachine* should not include leading backslashes (\).

TargetMachine
Specifies the computer to which the message will be sent. *TargetMachine* should not include leading backslashes (\).

Sender
Specifies the sender of the message. It is recommended that *Sender* be identical to *SendingMachine*, since your network configuration may refuse to send the message otherwise. When the message is displayed, this string will be identified as the sender of the message.

Message
Specifies the message itself. All text after the *Sender* parameter will be treated as part of *Message*, including spaces and quotation marks, although a **semicolon** will terminate *Message* and begin a new command.

DLL

Windows 2000	Ext.dll
Windows XP and later	Ext.dll

!obja

8/26/2019 • 2 minutes to read • Edit Online

The **!obja** extension displays the attributes of an object in the object manager.

```
!obja Address
```

Parameters

Address
Specifies the hexadecimal address of the object header you want to examine.

DLL

Windows 2000	Ext.dll Kdextx86.dll
Windows XP and later	Ext.dll

Additional Information

For information about objects and the object manager, see the Microsoft Windows SDK documentation, the Windows Driver Kit (WDK) documentation, and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

The attributes pertaining to the specified object are listed. Valid attributes are:

```
#define OBJ_INHERIT             0x00000002L
#define OBJ_PERMANENT           0x00000010L
#define OBJ_EXCLUSIVE           0x00000020L
#define OBJ_CASE_INSENSITIVE    0x00000040L
#define OBJ_OPENIF              0x00000080L
#define OBJ_OPENLINK            0x00000100L
#define OBJ_VALID_ATTRIBUTES    0x000001F2L
```

Here is an example:

```
kd> !obja 80967768
Obja +80967768 at 80967768:
        OBJ_INHERIT
        OBJ_PERMANENT
        OBJ_EXCLUSIVE
```

!owner

12/5/2018 • 2 minutes to read • Edit Online

The **!owner** extension displays the owner of a module or function.

```
!owner [Module[!Symbol]]
```

Parameters

Module
Specifies the module whose owner is desired. An asterisk (*) at the end of *Module* represents any number of additional characters.

Symbol
Specifies the symbol within *Module* whose owner is desired. An asterisk (*) at the end of *Symbol* represents any number of additional characters. If *Symbol* is omitted, the owner of the entire module is displayed.

DLL

Windows 2000	Ext.dll
Windows XP and later	Ext.dll

Remarks

If no parameters are used and a fault has occurred, **!owner** will display the name of the owner of the faulting module or function.

When you pass a module or function name to the **!owner** extension, the debugger displays the word **Followup** followed by the name of owner of the specified module or function.

For this extension to display useful information, you must first create a triage.ini file containing the names of the module and function owners.

For details on the triage.ini file and an example of the **!owner** extension, see Specifying Module and Function Owners.

!peb

4/8/2019 • 2 minutes to read • Edit Online

The **!peb** extension displays a formatted view of the information in the process environment block (PEB).

```
!peb [PEB-Address]
```

Parameters

PEB-Address
The hexadecimal address of the process whose PEB you want to examine. (This is not the address of the PEB as derived from the kernel process block for the process.) If *PEB-Address* is omitted in user mode, the PEB for the current process is used. If it is omitted in kernel mode, the PEB corresponding to the current process context is displayed.

DLL

Windows 2000	Kdextx86.dll Ntsdexts.dll
Windows XP and later	Exts.dll

Additional Information

For information about process environment blocks, see *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

The PEB is the user-mode portion of Microsoft Windows process control structures.

If the **!peb** extension with no argument gives you an error in kernel mode, you should use the **!process** extension to determine the PEB address for the desired process. Make sure your process context is set to the desired process, and then use the PEB address as the argument for **!peb**.

The exact output displayed depends on the Windows version and on whether you are debugging in kernel mode or user mode. The following example is taken from a kernel debugger attached to a Windows Server 2003 target:

```
kd> !peb
PEB at 7ffdf000
    InheritedAddressSpace:    No
    ReadImageFileExecOptions: No
    BeingDebugged:            No
    ImageBaseAddress:         4ad00000
    Ldr                       77fbe900
    Ldr.Initialized:          Yes
    Ldr.InInitializationOrderModuleList: 00241ef8 . 00242360
    Ldr.InLoadOrderModuleList:           00241e90 . 00242350
    Ldr.InMemoryOrderModuleList:         00241e98 . 00242358
            Base TimeStamp                     Module
        4ad00000 3d34633c Jul 16 11:17:32 2002 D:\WINDOWS\system32\cmd.exe
        77f40000 3d346214 Jul 16 11:12:36 2002 D:\WINDOWS\system32\ntdll.dll
        77e50000 3d3484ef Jul 16 13:41:19 2002 D:\WINDOWS\system32\kernel32.dll
....
    SubSystemData:     00000000
    ProcessHeap:       00140000
    ProcessParameters: 00020000
    WindowTitle:  'D:\Documents and Settings\Administrator\Desktop\Debuggers.lnk'
    ImageFile:    'D:\WINDOWS\system32\cmd.exe'
    CommandLine:  '"D:\WINDOWS\system32\cmd.exe" '
    DllPath:      'D:\WINDOWS\system32;D:\WINDOWS\system32;....
    Environment:  00010000
        ALLUSERSPROFILE=D:\Documents and Settings\All Users
        APPDATA=D:\Documents and Settings\UserTwo\Application Data
        CLIENTNAME=Console
....
        windir=D:\WINDOWS
```

The similar **!teb** extension displays the thread environment block.

!rebase

12/5/2018 • 2 minutes to read • Edit Online

The **!rebase** extension searches in a rebase.log file for a specified address or symbol.

```
!rebase [-r] Address [Path]
!rebase Symbol [Path]
!rebase -stack [Path]
!rebase -?
```

Parameters

-r
Attempts to load any module found in rebase.log.

Address
Specifies an address in standard hexadecimal format. The extension will search for DLLs near this address.

Path
Specifies the file path to the rebase.log. If *Path* is not specified, then the extension tries to guess the path to the rebase.log or, failing that, tries to read a rebase.log file from the current working directory.

Symbol
Specifies the symbol or image name. The extension will search for DLLs that contain this substring.

-stack
Displays all modules in the current stack.

-?
Displays a brief help text for this extension in the Debugger Command window.

DLL

Windows 2000	Unavailable
Windows XP and later	Ext.dll

!rtlavl

12/5/2018 • 2 minutes to read • Edit Online

The **!rtlavl** extension displays the entries of an RTL_AVL_TABLE structure.

```
!rtlavl Address [Module!Type]
!rtlavl -?
```

Parameters

Address
Specifies the address of the RTL_AVL_TABLE to display.

Module
Specifies the module in which the data structure is defined.

Type
Specifies the name of a data structure.

-?
Displays some brief Help text for this extension in the Debugger Command window.

DLL

Windows 2000	Ext.dll
Windows XP and later	Ext.dll

Additional Information

Use the **!gentable** extension to display AVL tables.

Remarks

Including the *Module***!***Type* option causes each entry in the table to be interpreted as having the given type.

The display can be interrupted at any time by pressing CTRL+BREAK (in WinDbg) or CTRL+C (in KD or CDB).

!sd

12/5/2018 • 2 minutes to read • Edit Online

The **!sd** extension displays the security descriptor at the specified address.

Syntax

```
!sd Address [Flags]
```

Parameters

Address
Specifies the hexadecimal address of the SECURITY_DESCRIPTOR structure.

Flags
If this is set to 1, the friendly name is displayed. This includes the security identifier (SID) type, as well as the domain and user name for the SID.

DLL

Exts.dll

Additional Information

For an application and an example of this command, see Determining the ACL of an Object. For information about security descriptors, see the Microsoft Windows SDK documentation, the Windows Driver Kit (WDK) documentation, and *Microsoft Windows Internals* by Mark Russinovich and David Solomon. Also see **!sid** and **!acl**.

Remarks

Here is an example:

```
kd> !sd e1a96a80 1
->Revision: 0x1
->Sbz1    : 0x0
->Control : 0x8004
            SE_DACL_PRESENT
            SE_SELF_RELATIVE
->Owner   : S-1-5-21-518066528-515770016-299552555-2981724 (User: MYDOMAIN\myuser)
->Group   : S-1-5-21-518066528-515770016-299552555-513 (Group: MYDOMAIN\Domain Users)
->Dacl    :
->Dacl    : ->AclRevision: 0x2
->Dacl    : ->Sbz1       : 0x0
->Dacl    : ->AclSize    : 0x40
->Dacl    : ->AceCount   : 0x2
->Dacl    : ->Sbz2       : 0x0
->Dacl    : ->Ace[0]: ->AceType: ACCESS_ALLOWED_ACE_TYPE
->Dacl    : ->Ace[0]: ->AceFlags: 0x0
->Dacl    : ->Ace[0]: ->AceSize: 0x24
->Dacl    : ->Ace[0]: ->Mask : 0x001f0003
->Dacl    : ->Ace[0]: ->SID: S-1-5-21-518066528-515770016-299552555-2981724 (User: MYDOMAIN\myuser)

->Dacl    : ->Ace[1]: ->AceType: ACCESS_ALLOWED_ACE_TYPE
->Dacl    : ->Ace[1]: ->AceFlags: 0x0
->Dacl    : ->Ace[1]: ->AceSize: 0x14
->Dacl    : ->Ace[1]: ->Mask : 0x001f0003
->Dacl    : ->Ace[1]: ->SID: S-1-5-18 (Well Known Group: NT AUTHORITY\SYSTEM)

->Sacl    :  is NULL
```

!sid

12/5/2018 • 2 minutes to read • Edit Online

The **!sid** extension displays the security identifier (SID) at the specified address.

Syntax

```
!sid Address [Flags]
```

Parameters

Address
Specifies the address of the SID structure.

Flags
If this is set to 1, the SID type, domain, and user name for the SID is displayed.

If this is set to 1, the friendly name is displayed. This includes the SID type, as well as the domain and user name for the SID.

DLL

Exts.dll

Additional Information

For information about SIDs, see the Microsoft Windows SDK documentation, the Windows Driver Kit (WDK) documentation, or *Microsoft Windows Internals* by Mark Russinovich and David Solomon. Also see **!sd** and **!acl**.

Remarks

Here are two examples, one without the friendly name shown, and one with:

```
kd> !sid 0xe1bf35b8
SID is: S-1-5-21-518066528-515770016-299552555-513

kd> !sid 0xe1bf35b8 1
SID is: S-1-5-21-518066528-515770016-299552555-513 (Group: MYGROUP\Domain Users)
```

!slist

12/5/2018 • 2 minutes to read • Edit Online

The **!slist** extension displays a singly-linked list (SList).

```
!slist Address [ Symbol [Offset] ]
!slist -?
```

Parameters

Address
Specifies the address of the SLIST_HEADER.

Symbol
Specifies the data type to use for display. If *Symbol* is specified, the debugger will assume that each node of the SList is an instance of this data type when displaying it.

Offset
Specifies the byte offset of the SList pointer within the structure.

-?
Displays some brief Help text for this extension in the Debugger Command window.

DLL

Windows 2000	Unavailable
Windows XP and later	Exts.dll

Remarks

If you know the nature of the linked structures, the *Symbol* and *Offset* parameters are very useful. To see the difference, here are two examples; the first omits the *Symbol* and *Offset* parameters, while the second includes them.

```
0:000> !slist ListHead
SLIST HEADER:
   +0x000 Alignment          : a000a002643e8
   +0x000 Next               : 2643e8
   +0x004 Depth              : a
   +0x006 Sequence           : a

SLIST CONTENTS:
002643e8  002642c0 0000000a 6e676953 72757461
002642c0  00264198 00000009 6e676953 72757461
00264198  00264070 00000008 6e676953 72757461
00264070  00263f48 00000007 6e676953 72757461
00263f48  00261420 00000006 6e676953 72757461
00261420  002612f8 00000005 6e676953 72757461
002612f8  002611d0 00000004 6e676953 72757461
002611d0  002610a8 00000003 6e676953 72757461
002610a8  00260f80 00000002 6e676953 72757461
00260f80  00000000 00000001 6e676953 72757461
```

```
0:000> !slist ListHead _PROGRAM_ITEM 0
SLIST HEADER:
 +0x000 Alignment          : a000a002643e8
   +0x000 Next               : 2643e8
 +0x004 Depth              : a
   +0x006 Sequence           : a

SLIST CONTENTS:
002643e8
   +0x000 ItemEntry        : _SINGLE_LIST_ENTRY
   +0x004 Signature        : 0xa
   +0x008 Description      : [260]  "Signature is: 10"
002642c0
   +0x000 ItemEntry        : _SINGLE_LIST_ENTRY
   +0x004 Signature        : 9
   +0x008 Description      : [260]  "Signature is: 9"
00264198
   +0x000 ItemEntry        : _SINGLE_LIST_ENTRY
   +0x004 Signature        : 8
   +0x008 Description      : [260]  "Signature is: 8"
00264070
   +0x000 ItemEntry        : _SINGLE_LIST_ENTRY
   +0x004 Signature        : 7
   +0x008 Description      : [260]  "Signature is: 7"
00263f48
   +0x000 ItemEntry        : _SINGLE_LIST_ENTRY
   +0x004 Signature        : 6
   +0x008 Description      : [260]  "Signature is: 6"
00261420
   +0x000 ItemEntry        : _SINGLE_LIST_ENTRY
   +0x004 Signature        : 5
   +0x008 Description      : [260]  "Signature is: 5"
002612f8
   +0x000 ItemEntry        : _SINGLE_LIST_ENTRY
   +0x004 Signature        : 4
   +0x008 Description      : [260]  "Signature is: 4"
002611d0
   +0x000 ItemEntry        : _SINGLE_LIST_ENTRY
   +0x004 Signature        : 3
   +0x008 Description      : [260]  "Signature is: 3"
002610a8
   +0x000 ItemEntry        : _SINGLE_LIST_ENTRY
   +0x004 Signature        : 2
   +0x008 Description      : [260]  "Signature is: 2"
00260f80
   +0x000 ItemEntry        : _SINGLE_LIST_ENTRY
   +0x004 Signature        : 1
   +0x008 Description      : [260]  "Signature is: 1"
```

!std_map

12/5/2018 • 2 minutes to read • Edit Online

The **!std_map** extension displays the entries of a std::map tree.

```
!std_map Address [Module!Type [TypeSize]]
!std_map -?
```

Parameters

Address
Specifies the address of the std::map tree to display.

Module
Specifies the module in which the data structure is defined.

Type
Specifies the name of a data structure. This must be expressed in *Module***!std::pair<***Type1***,***Type2***>** form. If the *TypeSize* parameter is used, this parameter must be enclosed in quotation marks.

TypeSize
Specifies the size of the data structure to make the symbols unambiguous.

-?
Displays some brief Help text for this extension in the Debugger Command window.

DLL

Windows 2000	Ext.dll
Windows XP and later	Ext.dll

Additional Information

To display other Standard Template Library (STL) defined templates, see **!stl**.

Remarks

Including the *Module***!***Type* option causes each entry in the table to be interpreted as having the given type.

Use **dt -ve (Module!std::pair<Type1,Type2>)** to display possible sizes.

!stl

12/5/2018 • 2 minutes to read • Edit Online

The **!stl** extension displays some of the known Standard Template Library (STL) templates.

```
!stl [Options] Template
!stl -?
```

Parameters

Options
May include any of the following possibilities:

-v
Causes detailed output to be displayed.

-V
Causes more detailed output to be displayed, such as information on the progress of the extension, including when certain functions are called and returned.

Template
Specifies the name of the template to be displayed.

-?
Displays some brief Help text for this extension in the Debugger Command window.

DLL

Windows 2000	Unavailable
Windows XP and later	Exts.dll

Remarks

The verbose options will only take effect if the debugger's verbose mode is enabled.

This extension currently supports STL templates of the following types: string, wstring, vector<*string*>, vector<*wstring*>, list<*string*>, list<*wstring*>, and pointers to any of the preceding types.

!str

12/5/2018 • 2 minutes to read • Edit Online

The **!str** extension displays an ANSI_STRING or OEM_STRING structure.

```
!str Address
```

Parameters

Address
Specifies the hexadecimal address of the ANSI_STRING or OEM_STRING structure.

DLL

Windows 2000	Ext.dll
Windows XP and later	Ext.dll

Additional Information

For more information about ANSI_STRING structures, see the Microsoft Windows SDK documentation.

Remarks

ANSI strings are counted 8-bit character strings, as defined in the following structure:

```
typedef struct _STRING {
    USHORT Length;
    USHORT MaximumLength;
    PCHAR Buffer;
} STRING;
typedef STRING ANSI_STRING;
typedef STRING OEM_STRING;
```

If the string is null-terminated, **Length** does not include the trailing null.

!sym

12/5/2018 • 2 minutes to read • Edit Online

The **!sym** extension controls noisy symbol loading and symbol prompts.

```
!sym
!sym noisy
!sym quiet
!sym prompts
!sym prompts off
```

Parameters

noisy
Activates noisy symbol loading.

quiet
Deactivates noisy symbol loading.

prompts
Allows authentication dialog boxes to appear when SymSrv receives an authentication request.

prompts off
Suppresses all authentication dialog boxes when SymSrv receives an authentication request. This may result in SymSrv being unable to access symbols over the internet.

DLL

Windows 2000	Dbghelp.dll
Windows XP and later	Dbghelp.dll

Remarks

If the **!sym** extension is used with no arguments, the current state of noisy symbol loading and symbol prompting is displayed.

The **!sym noisy** and **!sym quiet** extensions control noisy symbol loading. For details and for other methods of displaying and changing this option, see SYMOPT_DEBUG.

The **!sym prompts** and **!sym prompts off** extensions control whether authentication dialogs are displayed when SymSrv encounters an authentication request. These commands must be followed by **.reload (Reload Module)** for them to take effect. Authentication requests may be sent by proxy servers, internet firewalls, smart card readers, and secure websites. For details and for other methods of changing this option, see Firewalls and Proxy Servers.

Note Noisy symbol loading should not be confused with noisy source loading -- that is controlled by the **.srcnoisy (Noisy Source Loading)** command.

!symsrv

12/5/2018 • 2 minutes to read • Edit Online

The **!symsrv** extension closes the symbol server client.

```
!symsrv close
```

DLL

Windows 2000	Dbghelp.dll
Windows XP and later	Dbghelp.dll

Remarks

The **!symsrv close** extension will close any active symbol server client.

This can be useful if you need to re-synchronize your connection.

If you have previously refused an internet authentication request, you will need to use **!symsrv close** to reconnect to the symbol store. See Firewalls and Proxy Servers for details.

!teb

12/5/2018 • 2 minutes to read • Edit Online

The **!teb** extension displays a formatted view of the information in the thread environment block (TEB).

```
!teb [TEB-Address]
```

Parameters

TEB-Address
The hexadecimal address of the thread whose TEB you want to examine. (This is not the address of the TEB as derived from the kernel thread block for the thread.) If *TEB-Address* is omitted in user mode, the TEB for the current thread is used. If it is omitted in kernel mode, the TEB corresponding to the current register context is displayed.

DLL

Exts.dll

Additional Information

For information about thread environment blocks, see *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

The TEB is the user-mode portion of Microsoft Windows thread control structures.

If the **!teb** extension with no argument gives you an error in kernel mode, you should use the **!process** extension to determine the TEB address for the desired thread. Make sure your register context is set to the desired thread, and then use the TEB address as the argument for **!teb**.

Here is an example of this command's output in user mode:

```
0:001> ~
   0  id: 324.458   Suspend: 1 Teb 7ffde000 Unfrozen
.  1  id: 324.48c   Suspend: 1 Teb 7ffdd000 Unfrozen

0:001> !teb
TEB at 7FFDD000
    ExceptionList:    76ffdc
    Stack Base:       770000
    Stack Limit:      76f000
    SubSystemTib:     0
 FiberData:        1e00
    ArbitraryUser:    0
    Self:             7ffdd000
    EnvironmentPtr:   0
 ClientId:         324.48c
    Real ClientId:    324.48c
    RpcHandle:        0
    Tls Storage:      0
    PEB Address:      7ffdf000
    LastErrorValue:   0
    LastStatusValue:  0
    Count Owned Locks:0
    HardErrorsMode:   0
```

The similar **!peb** extension displays the process environment block.

!tls

12/5/2018 • 2 minutes to read • Edit Online

The **!tls** extension displays a thread local storage (TLS) slot.

```
!tls Slot [TEB]
```

Parameters

Slot
Specifies the TLS slot. This can be any value between 0 and 1088 (decimal). If *Slot* is -1, all slots are displayed.

TEB
Specifies the thread environment block (TEB). If this is 0 or omitted, the current thread is used.

DLL

Windows 2000	Unavailable
Windows XP and later	Exts.dll

Remarks

Here is an example:

```
0:000> !tls -1
TLS slots on thread: c08.f54
0x0000 : 00000000
0x0001 : 003967b8
0:000> !tls 0
c08.f54: 00000000
```

!token

12/5/2018 • 3 minutes to read • Edit Online

The **!token** extension displays a formatted view of a security token object.

Kernel-Mode Syntax:

```
!token [-n] [Address]
!token -?
```

User-Mode Syntax:

```
!token [-n] [Handle]
!token -?
```

Parameters

Address
(Kernel mode only) Specifies the address of the token to be displayed. If this is 0 or omitted, the token for the active thread is displayed.

Handle
(User mode only) Specifies the handle of the token to be displayed. If this is 0 or omitted, the token associated with the target thread is displayed.

-n
(User mode only) Causes the display to include the friendly name. This includes the security identifier (SID) type, as well as the domain and user name for the SID. This option cannot be used when you are debugging LSASS.

-?
Displays help text for this extension.

DLL

Exts.dll

The **!token** command is available in kernel-mode and live user-mode debugging. It cannot be used on user-mode dump files.

Additional Information

For information about the kernel-mode TOKEN structure, see *Microsoft Windows Internals* by Mark Russinovich and David Solomon. For information about the user-mode TOKEN structure, see the Microsoft Windows SDK documentation.

Remarks

The TOKEN structure is a security object type that represents an authenticated user process. Every process has an assigned token, which becomes the default token for each thread of that process. However, an individual thread can be assigned a token that overrides this default.

You can get the token address from the output of **!process**. To display a list of the individual fields of the TOKEN structure, use the **dt nt!_TOKEN** command.

Here is an example:

```
kd> !process 81464da8 1
PROCESS 81464da8  SessionId: 0  Cid: 03bc    Peb: 7ffdf000  ParentCid: 0124
    DirBase: 0dec2000  ObjectTable: e1a31198  TableSize: 275.
    Image: MSMSGS.EXE
    VadRoot 81468cc0 Vads 170 Clone 0 Private 455. Modified 413. Locked 0.
    DeviceMap e1958438
    Token                             e1bed030
    ElapsedTime                       0:44:15.0142
    UserTime                          0:00:00.0290
    KernelTime                        0:00:00.0300
    QuotaPoolUsage[PagedPool]         49552
    QuotaPoolUsage[NonPagedPool]      10872
    Working Set Sizes (now,min,max)  (781, 50, 345) (3124KB, 200KB, 1380KB)
    PeakWorkingSetSize                1550
    VirtualSize                       57 Mb
    PeakVirtualSize                   57 Mb
    PageFaultCount                    2481
    MemoryPriority                    BACKGROUND
    BasePriority                      8
    CommitCharge                      2497
kd> !exts.token -n e1bed030
_TOKEN e1bed030
TS Session ID: 0
User: S-1-5-21-518066528-515770016-299552555-2981724 (User: MYDOMAIN\myuser)
Groups:
 00 S-1-5-21-518066528-515770016-299552555-513 (Group: MYDOMAIN\Domain Users)
    Attributes - Mandatory Default Enabled
 01 S-1-1-0 (Well Known Group: localhost\Everyone)
    Attributes - Mandatory Default Enabled
 02 S-1-5-32-544 (Alias: BUILTIN\Administrators)
    Attributes - Mandatory Default Enabled Owner
 03 S-1-5-32-545 (Alias: BUILTIN\Users)
    Attributes - Mandatory Default Enabled
 04 S-1-5-21-518066528-515770016-299552555-2999049 (Group: MYDOMAIN\AllUsers)
    Attributes - Mandatory Default Enabled
 05 S-1-5-21-518066528-515770016-299552555-2931095 (Group: MYDOMAIN\SomeGroup1)
    Attributes - Mandatory Default Enabled
 06 S-1-5-21-518066528-515770016-299552555-2931096 (Group: MYDOMAIN\SomeGroup2)
    Attributes - Mandatory Default Enabled
 07 S-1-5-21-518066528-515770016-299552555-3014318 (Group: MYDOMAIN\SomeGroup3)
    Attributes - Mandatory Default Enabled
 08 S-1-5-21-518066528-515770016-299552555-3053352 (Group: MYDOMAIN\Another Group)
    Attributes - Mandatory Default Enabled
 09 S-1-5-21-518066528-515770016-299552555-2966661 (Group: MYDOMAIN\TestGroup)
    Attributes - Mandatory Default Enabled
 10 S-1-5-21-2117033040-537160606-1609722162-17637 (Group: MYOTHERDOMAIN\someusers)
    Attributes - Mandatory Default Enabled
 11 S-1-5-21-518066528-515770016-299552555-3018354 (Group: MYDOMAIN\TestGroup2)
    Attributes - Mandatory Default Enabled
 12 S-1-5-21-518066528-515770016-299552555-3026602 (Group: MYDOMAIN\SomeGroup4)
    Attributes - Mandatory Default Enabled
 13 S-1-5-21-518066528-515770016-299552555-2926570 (Group: MYDOMAIN\YetAnotherGroup)
    Attributes - Mandatory Default Enabled
 14 S-1-5-21-661411660-2927047998-133698966-513 (Group: MYDOMAIN\Domain Users)
    Attributes - Mandatory Default Enabled
 15 S-1-5-21-518066528-515770016-299552555-2986081 (Alias: MYDOMAIN\an_alias)
    Attributes - Mandatory Default Enabled GroupResource
 16 S-1-5-21-518066528-515770016-299552555-3037986 (Alias: MYDOMAIN\AReallyLongGroupName1)
    Attributes - Mandatory Default Enabled GroupResource
 17 S-1-5-21-518066528-515770016-299552555-3038991 (Alias: MYDOMAIN\AReallyLongGroupName2)
    Attributes - Mandatory Default Enabled GroupResource
 18 S-1-5-21-518066528-515770016-299552555-3037999 (Alias: MYDOMAIN\AReallyLongGroupName3)
    Attributes - Mandatory Default Enabled GroupResource
 19 S-1-5-21-518066528-515770016-299552555-3038983 (Alias: MYDOMAIN\AReallyReallyLongGroupName)
    Attributes - Mandatory Default Enabled GroupResource
 20 S-1-5-5-0-71188 (no name mapped)
```

```
    Attributes - Mandatory Default Enabled LogonId
 21 S-1-2-0 (Well Known Group: localhost\LOCAL)
    Attributes - Mandatory Default Enabled
 22 S-1-5-4 (Well Known Group: NT AUTHORITY\INTERACTIVE)
    Attributes - Mandatory Default Enabled
 23 S-1-5-11 (Well Known Group: NT AUTHORITY\Authenticated Users)
    Attributes - Mandatory Default Enabled
Primary Group: S-1-5-21-518066528-515770016-299552555-513 (Group: MYDOMAIN\Domain Users)
Privs:
 00 0x000000017 SeChangeNotifyPrivilege           Attributes - Enabled Default
 01 0x000000008 SeSecurityPrivilege               Attributes -
 02 0x000000011 SeBackupPrivilege                 Attributes -
 03 0x000000012 SeRestorePrivilege                Attributes -
 04 0x00000000c SeSystemtimePrivilege             Attributes -
 05 0x000000013 SeShutdownPrivilege               Attributes -
 06 0x000000018 SeRemoteShutdownPrivilege         Attributes -
 07 0x000000009 SeTakeOwnershipPrivilege          Attributes -
 08 0x000000014 SeDebugPrivilege                  Attributes -
 09 0x000000016 SeSystemEnvironmentPrivilege      Attributes -
 10 0x00000000b SeSystemProfilePrivilege          Attributes -
 11 0x00000000d SeProfileSingleProcessPrivilege   Attributes -
 12 0x00000000e SeIncreaseBasePriorityPrivilege   Attributes -
 13 0x00000000a SeLoadDriverPrivilege             Attributes - Enabled
 14 0x00000000f SeCreatePagefilePrivilege         Attributes -
 15 0x000000005 SeIncreaseQuotaPrivilege          Attributes -
 16 0x000000019 SeUndockPrivilege                 Attributes - Enabled
 17 0x00000001c SeManageVolumePrivilege           Attributes -
Authentication ID:         (0,11691)
Impersonation Level:       Anonymous
TokenType:                 Primary
Source: User32             TokenFlags: 0x9 ( Token in use )
Token ID: 18296            ParentToken ID: 0
Modified ID:               (0, 18298)
RestrictedSidCount: 0      RestrictedSids: 00000000
```

!tp

12/5/2018 • 2 minutes to read • Edit Online

The **!tp** extension displays thread pool information.

```
!tp pool Address [Flags]
!tp tqueue Address [Flags]
!tp ItemType Address [Flags]
!tp ThreadType [Address]
!tp stats Address [Flags]
!tp wfac Address
!tp wqueue Address Priority Node
!tp -?
```

Parameters

pool **** *Address*
Causes the entire thread pool at *Address* to be displayed. If *Address* is 0, then all thread pools will be displayed.

tqueue **** *Address*
Causes the active timer queue at *Address* to be displayed.

ItemType Address
Causes the specified thread pool item to be displayed. *Address* specifies the address of the item. *ItemType* specifies the type of the item; this can include any of the following possibilities:

obj
A generic pool item (such as an IO item) will be displayed.

timer
A timer item will be displayed.

wait
A wait item will be displayed.

work
A work item will be displayed.

ThreadType [*Address*]
Causes threads of the specified type to be displayed. If *Address* is included and nonzero, then only the thread at this address is displayed. If *Address* is 0, all threads matching *ThreadType* are displayed. If *Address* is omitted, only the threads matching *ThreadType* associated with the current thread are displayed. *ThreadType* specifies the type of the thread to be displayed; this can include any of the following possibilities:

waiter
A thread pool waiter thread will be displayed.

worker
A thread pool worker thread will be displayed.

stats [*Address***]**
Causes the debug statistics of the current thread to be displayed. *Address* may be omitted, but if it is specified, it must equal -1 (negative one), to represent the current thread.

wfac **** *Address*
(Windows 7 and later only) Causes the worker factory at *Address* to be displayed. The specified *Address* must be a valid nonzero address.

wqueue **** *Address*
(Windows 7 and later only) Causes display of a work queue and NUMA node that matche the following: a specified priority, a specified NUMA node, and the pool, at a specified address, to which the NUMA node belongs. *Address* specifies the address of the pool. When the **wqueue** parameter is used, it must be followed by *Address*, *Priority*, and *Node*.

Priority
(Windows 7 and later only) Specifies the priority levels of the work queues to be displayed. *Priority* can be any of the following values:

0
Work queues with high priority are displayed.

1
Work queues with normal priority are displayed.

2
Work queues with low priority are displayed.

-1
All work queues are displayed.

Node
(Windows 7 and later only) Specifies a NUMA node belonging to the pool specified by *Address*. If *Node* is -1 (negative one), all NUMA nodes are displayed.

Flags
Specifies what the display should contain. This can be a sum of any of the following bit values (the default is 0x0):

Bit 0 (0x1)
Causes the display to be single-line output. This bit value has no effect on the output when an *ItemType* is displayed.

Bit 1 (0x2)
Causes the display to include member information.

Bit 2 (0x4)
This flag is relevant only when the **pool** option is used. In Windows XP, Windows Server 2003, Windows Vista, and Windows Server 2008, this flag causes the display to include the pool work queue. In Windows 7 and later, this flag causes the display to include all the pool's work queues that are at normal priority, and all NUMA nodes.

-?
Displays a brief help text for this extension in the Debugger Command window.

DLL

Windows 2000	Unavailable
Windows XP and later	Exts.dll

Additional Information

For information about thread pooling, see the Microsoft Windows SDK documentation.

!triage

12/5/2018 • 2 minutes to read • Edit Online

The **!triage** extension command is obsolete. Use **!analyze** instead.

!ustr

12/5/2018 • 2 minutes to read • Edit Online

The **!ustr** extension displays a UNICODE_STRING structure.

```
!ustr Address
```

Parameters

Address
Specifies the hexadecimal address of the UNICODE_STRING structure.

DLL

Windows 2000	Ext.dll
Windows XP and later	Ext.dll

Additional Information

For more information about UNICODE_STRING structures, see the Microsoft Windows SDK documentation.

Remarks

Unicode strings are counted 16-bit character strings, as defined in the following structure:

```
typedef struct _UNICODE_STRING {
    USHORT Length;
    USHORT MaximumLength;
    PWSTR  Buffer;
} UNICODE_STRING;
```

If the string is null-terminated, **Length** does not include the trailing null.

Most Win32 character string arguments are converted to Unicode strings before any real work is done.

!version

12/5/2018 • 2 minutes to read • Edit Online

The **!version** extension displays the version information for the extension DLL.

This extension command should not be confused with the **version (Show Debugger Version)** command.

```
![ExtensionDLL.]version
```

Parameters

ExtensionDLL
Specifies the extension DLL whose version number is to be displayed.

DLL

This extension is available in most extension DLLs.

Remarks

If the extension DLL version does not match the debugger version, error messages will be displayed.

This extension command will not work on Windows XP and later versions of Windows. To display version information, use the **version (Show Debugger Version)** command.

The original purpose of this extension was to ensure that the DLL version matched the target version, since a mismatch would result in inaccurate results for many extensions. Newer DLLs are no longer restricted to working with only one version of Windows, so this extension is obsolete.

!winrterr

12/5/2018 • 2 minutes to read • Edit Online

The **!winrterr** sets the debugger reporting mode for Windows Runtime errors.

```
!winrterr Mode
!winrterr
```

Parameters

Mode

The following table describes the possible values for *Mode*.

VALUE	DESCRIPTION
report	When a Windows Runtime error occurs, the error and related text are displayed in the debugger, but execution contunues. This is the default mode.
break	When a Windows Runtime error occurs, the error and related text are displayed in the debugger, and execution stops.
quiet	When a Windows Runtime error occurs, nothing is displayed in the debugger, and execution continues.

If *Mode* is omitted, **!winrterr** displays the current reporting mode. If the debugger has broken in as a result of a Windows Runtime error, the error and related text are also displayed.

See also

Windows Runtime Debugger Commands

!hstring

!hstring2

Kernel-Mode Extensions

12/5/2018 • 2 minutes to read • Edit Online

This section of the reference describes extension commands that are primarily used during kernel-mode debugging.

The debugger will automatically load the proper version of these extension commands. Unless you have manually loaded a different version, you do not have to keep track of the DLL versions being used. See Using Debugger Extension Commands for a description of the default module search order. See Loading Debugger Extension DLLs for an explanation of how to load extension modules.

Each extension command reference page lists the DLLs that expose that command. Use the following rules to determine the proper directory from which to load this extension DLL:

- If your target computer is running Windows XP or a later version of Windows, use winxp\Kdexts.dll.

In addition, kernel-mode extensions that are not specific to any single operating system can be found in winext\kext.dll.

!ahcache

12/5/2018 • 2 minutes to read • Edit Online

The **!ahcache** extension displays the application compatibility cache.

```
!ahcache [Flags]
```

Parameters

Flags
Specifies the information to include in the display. This can be any combination of the following bits (the default is zero):

Bit 0 (0x1)
Displays the RTL_GENERIC_TABLE list instead of the LRU list.

Bit 4 (0x10)
Verbose display: includes all entry details, not just the names.

DLL

Windows 2000	Unavailable
Windows XP and later	Kdexts.dll

!alignmentfaults

12/5/2018 • 2 minutes to read • Edit Online

The **!alignmentfaults** extension displays all current type alignment faults by location and image, sorted by frequencies.

```
!alignmentfaults
```

DLL

Windows 2000	Unavailable
Windows XP and later	Kdexts.dll

Additional Information

For information about alignment faults, see the Microsoft Windows SDK documentation.

Remarks

This is only available on checked builds.

!analyzebugcheck

12/5/2018 • 2 minutes to read • Edit Online

The **!analyzebugcheck** extension command is obsolete. Use **!analyze** instead.

!apc

12/5/2018 • 2 minutes to read • Edit Online

The **!apc** extension formats and displays the contents of one or more asynchronous procedure calls (APCs).

```
!apc
!apc proc Process
!apc thre Thread
!apc KAPC
```

Parameters

Process
Specifies the address of the process whose APCs are to be displayed.

Thread
Specifies the address of the thread whose APCs are to be displayed.

KAPC
Specifies the address of the kernel APC to be displayed.

DLL

Windows 2000	Unavailable
Windows XP and later	Kdexts.dll

Additional Information

For information about APCs, see the Windows Driver Kit (WDK) documentation and Microsoft Windows Internals by Mark Russinovich and David Solomon.

Remarks

Without any parameters, **!apc** displays all APCs.

Here is an example:

```
kd> !apc
*** Enumerating APCs in all processes
Process e0000000858ba8b0 System
Process e0000165fff86040 smss.exe
Process e0000165fff8c040 csrss.exe
Process e0000165fff4e1d0 winlogon.exe
Process e0000165fff101d0 services.exe
Process e0000165fffa81d0 lsass.exe
Process e0000165fff201d0 svchost.exe
Process e0000165fff8e040 svchost.exe
Process e0000165fff3e040 svchost.exe
Process e0000165fff6e040 svchost.exe
Process e0000165fff24040 spoolsv.exe
Process e000000085666640 wmiprvse.exe
Process e00000008501e520 wmiprvse.exe
Process e0000000856db480 explorer.exe
Process e0000165fff206a0 ctfmon.exe
Process e0000000850009d0 ctfmon.exe
Process e0000165fff51600 conime.exe
Process e000000085496340 taskmgr.exe
Process e000000085489c30 userinit.exe
```

!apicerr

4/8/2019 • 2 minutes to read • Edit Online

The **!apicerr** extension displays the local Advanced Programmable Interrupt Controller (APIC) error log.

```
!apicerr [Format]
```

Parameters

Format
Specifies the order in which to display the error log contents. This can be any one of the following values:

0x0
Displays the error log according to order of occurrence.

0x1
Displays the error log according to processor.

DLL

Windows 2000	Unavailable
Windows XP and later	Kdexts.dll

This extension command can only be used with an x86-based or an x64-based target computer.

Additional Information

For information about APICs, see *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

!arbinst

12/5/2018 • 2 minutes to read • Edit Online

The **!arbinst** extension displays information about a specified arbiter.

```
!arbinst Address [Flags]
```

Parameters

Address
Specifies the hexadecimal address of the arbiter to be displayed.

Flags
Specifies how much information to display for each arbiter. At present, the only flag is 0x100. If this flag is set, then the aliases are displayed.

DLL

Windows 2000	Unavailable
Windows XP and later	Kdexts.dll

Additional Information

See also the **!arbiter** extension.

Remarks

For the arbiter specified, **!arbinst** displays each allocated range of system resources, some optional flags, the PDO attached to that range (in other words, the range's owner), and the service name of this owner (if known).

Here is an example:

```
kd> !arbinst e0000106002ee8e8
Port Arbiter "PCI I/O Port (b=02)" at e0000106002ee8e8
  Allocated ranges:
    0000000000000000 - 0000000000001fff       00000000 <Not on bus>
    0000000000002000 - 00000000000020ff     P e0000000858bea20  (ql1280)
    0000000000003000 - ffffffffffffffff       00000000 <Not on bus>
  Possible allocation:
    < none >
kd> !arbinst e0000106002ec458
Memory Arbiter "PCI Memory (b=02)" at e0000106002ec458
  Allocated ranges:
    0000000000000000 - 00000000ebffffff       00000000 <Not on bus>
    00000000effdef00 - 00000000effdefff   B   e0000000858be560
    00000000effdf000 - 00000000effdffff       e0000000858bea20  (ql1280)
    00000000f0000000 - ffffffffffffffff       00000000 <Not on bus>
  Possible allocation:
    < none >
```

!arbiter

12/5/2018 • 2 minutes to read • Edit Online

The **!arbiter** extension displays the current system resource arbiters and arbitrated ranges.

```
!arbiter [Flags]
```

Parameters

Flags
Specifies which classes of arbiters are displayed. If omitted, all arbiters are displayed. These bits can be combined freely.

Bit 0 (0x1)
Display I/O arbiters.

Bit 1 (0x2)
Display memory arbiters.

Bit 2 (0x4)
Display IRQ arbiters.

Bit 3 (0x8)
Display DMA arbiters.

Bit 4 (0x10)
Display bus number arbiters.

Bit 8 (0x100)
Do not display aliases.

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Kdexts.dll

Additional Information

See Plug and Play Debugging for applications of this extension command.

Remarks

For each arbiter, **!arbiter** displays each allocated range of system resources, some optional flags, the PDO attached to that range (in other words, the range's owner), and the service name of this owner (if known).

The flags have the following meanings:

FLAG	MEANING
S	Range is shared
C	Range in conflict
B	Range is boot-allocated
D	Range is driver-exclusive
A	Range alias
P	Range positive decode

Here is an example:

```
kd> !arbiter 4

DEVNODE 80e203b8 (HTREE\ROOT\0)
  Interrupt Arbiter "" at 80167140
    Allocated ranges:
      0000000000000000 - 0000000000000000   B   80e1d3d8
      0000000000000001 - 0000000000000001   B   80e1d3d8
 .....
      00000000000001a2 - 00000000000001a2
        00000000000001a2 - 00000000000001a2  CB   80e1d3d8
        00000000000001a2 - 00000000000001a2  CB   80e52538  (Serial)
      00000000000001a3 - 00000000000001a3       80e52778  (i8042prt)
      00000000000001b3 - 00000000000001b3       80e1b618  (i8042prt)
 Possible allocation:
      < none >
```

In this example, the next-to-last line shows the resource range (which consists of 0x1A3 alone), the PDO of 0x80E52778, and the service of i8042prt.sys. No flags are listed on this line.

You can now use **!devobj** with this PDO address to find the device extension and device node addresses:

```
kd> !devobj 80e52778
Device object (80e52778) is for:
 00000034 \Driver\PnpManager DriverObject 80e20610
Current Irp 00000000 RefCount 1 Type 00000004 Flags 00001040
DevExt 80e52830 DevObjExt 80e52838 DevNode 80e52628
ExtensionFlags (0000000000)
AttachedDevice (Upper) 80d78b28 \Driver\i8042prt
Device queue is not busy.
```

!ate

4/8/2019 • 2 minutes to read • Edit Online

The **!ate** extension displays the alternate page table entry (ATE) for the specified address.

```
!ate Address
```

Important This command has been deprecated in the Windows Debugger Version 10.0.14257 and later, and is no longer available.

Parameters

Address
Specifies the ATE to display.

DLL

Windows 2000	Unavailable
Windows XP and later	Kdexts.dll

Additional Information

For information about page tables and page directories, see *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

This extension is only available on Itanium-based computers.

The status flags for the ATE are shown in the following table. The **!ate** display indicates these bits with capital letters or dashes and adds additional information as well.

DISPLAY WHEN SET	DISPLAY WHEN CLEAR	MEANING
V	-	Commit.
G	-	Not accessed.
E	-	Execute.
W	R	Writeable or read-only.

DISPLAY WHEN SET	DISPLAY WHEN CLEAR	MEANING
L	-	Locked. The ATE is locked, therefore, any faults on the page that contain the ATE will be retried until the current fault is satisfied. This can happen on multi-processor systems.
Z	-	Fill zero.
N	-	No access.
C	-	Copy on Write.
I	-	PTE indirect. This ATE indirectly references another physical page. The page that contains the ATE might have two conflicting ATE attributes.
P		Reserved.

!bcb

8/26/2019 • 2 minutes to read • Edit Online

The **!bcb** extension displays the specified buffer control block.

```
!bcb Address
```

Parameters

Address
Specifies the address of the buffer control block.

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Unavailable (see Remarks section)

Additional Information

For information about cache management, see the Microsoft Windows SDK documentation and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

For information about other cache management extensions, use the **!cchelp** extension.

Remarks

This extension is available for Windows 2000 only. In Windows XP or later, use the **dt nt!_BCB Address** command to display the buffer control block directly.

!blockeddrv

12/5/2018 • 2 minutes to read • Edit Online

The **!blockeddrv** extension displays the list of blocked drivers on the target computer.

```
!blockeddrv
```

DLL

Windows 2000	Unavailable
Windows XP and later	Kdexts.dll

Remarks

Here is an example:

```
kd> !blockeddrv
Driver:      Status    GUID
afd.sys      0:        {00000008-0206-0001-0000-000030C964E1}
agp440.sys   0:        {0000005C-175A-E12D-5000-010020885580}
atapi.sys    0:        {0000005C-B04A-E12E-5600-000020885580}
audstub.sys  0:        {0000005C-B04A-E12E-5600-000020885580}
Beep.SYS     0:        {0000005C-B04A-E12E-5600-000020885580}
Cdfs.SYS     0:        {00000008-0206-0001-0000-000008F036E1}
.....
```

!bpid

12/5/2018 • 2 minutes to read • Edit Online

The **!bpid** extension requests that a process on the target computer break into the debugger or requests that a user-mode debugger be attached to a process on the target computer.

```
!bpid [Options] PID
```

Parameters

Option
Controls the additional activities of this command.

The valid values for *Option* appear in the following table.

-a	Attaches a new user-mode debugger to the process specified by *PID*. The user-mode debugger runs on the target machine.
-s	Adds a breakpoint that occurs in the WinLogon process immediately before the break in the user-mode process specified by *PID*. This allows the user to verify the request before attempting the action.
-w	Stores the request in the memory in the target computer. The target system can then repeat the request, but this is not usually necessary.

PID
Specifies the process ID of the desired process on the target computer. If you are using this to control a user-mode debugger on the target computer, *PID* should be the process ID of the target application, not of the user-mode debugger. (Because process IDs are usually listed in decimal format, you might need to prefix this with **0n** or convert it to hexadecimal format.)

DLL

Windows 2000	Unavailable
Windows XP and later	Kdexts.dll

This extension command is supported on x86-based, x64-based, and Itanium-based target computers.

Remarks

This command is especially useful when redirecting input and output from a user-mode debugger to the kernel debugger. It causes the user-mode target application to break into the user-mode debugger, which in turn requests input from the kernel debugger. See Controlling the User-Mode Debugger from the Kernel Debugger for details.

If this command is used in another situation, the user-mode process calls **DbgBreakPoint**. This will usually break directly into the kernel debugger.

The **-s** option causes a break in WinLogon just before the break in the specified process occurs. This is useful if you want to perform debugging actions within WinLogon's process context. The **g (Go)** command can then be used to move on to the second break.

Note that there are ways in which this extension can fail to execute:

- Lack of resources. The **!bpid** extension injects a thread into the target process, so the system must have enough resources to create a thread. Using the **-a** option requires even more system resources since **!bpid -a** must run a full instance of a debugger on the target computer.
- The loader lock is already held. Both **!bpid** and **!bpid -a** require a thread to run in the target process in order to make it break into the debugger. If another thread is holding the loader lock, then the **!bpid** thread will not be able to run and a break into the debugger may not occur. Thus, if **!bpid** fails when there is enough user-mode memory available for the target process, it is possible that the loader lock is held.
- Lack of permission. The operation of the !bpid extension requires permission sufficient for WinLogon to create a remote thread and attach a debugger to a given process.
- No access to ntsd.exe. If ntsd.exe is not found in a commonly known path, !bpid will fail to set an appropriate PID. Note that ntsd.exe is not included by default with Windows Vista.

!btb

12/5/2018 • 2 minutes to read • Edit Online

The **!btb** extension displays the Itanium-based processor, branch traces buffer (BTB) configuration and trace registers for the current processor.

```
!btb
```

Important This command has been deprecated in the Windows Debugger Version 10.0.14257 and later, and is no longer available.

DLL

Windows 2000	Unavailable
Windows XP and later	Kdexts.dll

This extension command can only be used with an Itanium-based target computer.

!bth

12/5/2018 • 2 minutes to read • Edit Online

The **!bth** extension displays the Itanium-based branch traces history for the specified processor.

```
!bth [Processor]
```

Important This command has been deprecated in the Windows Debugger Version 10.0.14257 and later, and is no longer available.

Parameters

Processor
Specifies a processor. If *Processor* is omitted, then the branch trace history for all of processors is displayed.

DLL

Windows 2000	Unavailable
Windows XP and later	Kdexts.dll

This extension command can only be used with an Itanium-based target computer.

!bugdump

10/23/2019 • 2 minutes to read • Edit Online

The **!bugdump** extension formats and displays the information contained in the bug check callback buffers.

```
!bugdump [Component]
```

Parameters

Component
Specifies the component whose callback data is to be examined. If omitted, all bug check callback data is displayed.

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Kdexts.dll

Additional Information

For more information, see Reading Bug Check Callback Data.

Remarks

This extension can only be used after a bug check has occurred, or when you are debugging a kernel-mode crash dump file.

The *Component* parameter corresponds to the final parameter used in **KeRegisterBugCheckCallback**.

The buffers that hold callback data are not available in a Small Memory Dump. These buffers are present in Kernel Memory Dumps and Full Memory Dumps. However, in Windows XP SP1, Windows Server 2003, and later versions of Windows, the dump file is created before the drivers' **BugCheckCallback** routines are called, and therefore these buffers will not contain the data written by these routines.

If you are performing live debugging of a crashed system, all callback data will be present.

!bushnd

12/5/2018 • 2 minutes to read • Edit Online

The **!bushnd** extension displays a HAL BUS_HANDLER structure.

```
!bushnd [Address]
```

Parameters

Address
Specifies the hexadecimal address of the HAL BUS_HANDLER structure. If omitted, **!bushnd** displays a list of buses and the base address of the handler.

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Kdexts.dll

!ca

4/8/2019 • 2 minutes to read • Edit Online

The **!ca** extension displays information about a control area.

```
!ca [Address | 0 | -1] [Flags]
```

Parameters

Address
Address of the control area. If you specify 0 for this parameter, information is displayed about all control areas. If you specify -1 for this parameter, information is displayed about the unused segment list.

Flags
Flags that specify which information is displayed. This parameter is a bitwise OR of one or more of the following flags.

FLAG	DESCRIPTION
0x1	Display segment information.
0x2	Display subsection information.
0x4	Display the list of mapped views. (Windows 7 and later)
0x8	Display compact (single line) output.
0x10	Display file-backed control areas.
0x20	Display control areas backed by the page file.
0x40	Display image control areas.

If none of the last three flags are specified, all three types of control area are displayed.

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Kdexts.dll

Additional Information

For information about control areas, see *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

To get a list of the control areas of all mapped files, use the **!memusage** extension.

Here is an example:

```
kd> !memusage
 loading PFN database
loading (99% complete)
             Zeroed:     16 (    64 kb)
               Free:      0 (     0 kb)
            Standby:   2642 ( 10568 kb)
           Modified:    720 (  2880 kb)
    ModifiedNoWrite:      0 (     0 kb)
       Active/Valid:  13005 ( 52020 kb)
         Transition:      0 (     0 kb)
            Unknown:      0 (     0 kb)
              TOTAL:  16383 ( 65532 kb)
  Building kernel map
  Finished building kernel map

  Usage Summary (in Kb):
Control Valid Standby Dirty Shared Locked PageTables  name
ff8636e8    56    376     0     0     0     0  mapped_file( browseui.dll )
ff8cf388    24      0     0     0     0     0  mapped_file( AVH32DLL.DLL )
ff8d62c8    12      0     0     0     0     0  mapped_file( PSAPI.DLL )
ff8dd468   156     28     0     0     0     0  mapped_file( INOJOBSV.EXE )
fe424808   136     88     0    52     0     0  mapped_file( oleaut32.dll )
fe4228a8   152     44     0   116     0     0  mapped_file( MSVCRT.DLL )
ff8ec848     4      0     0     0     0     0    No Name for File
ff859de8     0     32     0     0     0     0  mapped_file( timedate.cpl )
. . . . .

kd> !ca ff8636e8

ControlArea @ff8636e8
  Segment:    e1b74548    Flink              0   Blink:               0
 Section Ref        0    Pfn Ref           6c   Mapped Views:        1
  User Ref           1    Subsections        5   Flush Count:         0
  File Object ff86df88    ModWriteCount      0   System Views:        0
  WaitForDel         0    Paged Usage      380   NonPaged Usage       e0
  Flags (10000a0) Image File HadUserReference

   File: \WINNT\System32\browseui.dll

Segment @ e1b74548:
   Base address        0  Total Ptes        c8  NonExtendPtes:       c8
   Image commit        1  ControlArea ff8636e8  SizeOfSegment: c8000
   Image Base          0  Committed          0  PTE Template:   31b8438
 Based Addr   76e10000  ProtoPtes   e1b74580  Image Info:    e1b748a4

Subsection 1. @ ff863720
   ControlArea: ff8636e8  Starting Sector 0 Number Of Sectors 2
   Base Pte     e1b74580  Ptes In subsect        1 Unused Ptes          0
   Flags              15  Sector Offset          0 Protection           1
    ReadOnly CopyOnWrite

Subsection 2. @ ff863740
   ControlArea: ff8636e8  Starting Sector 2 Number Of Sectors 3d0
   Base Pte     e1b74584  Ptes In subsect       7a Unused Ptes          0
   Flags              35  Sector Offset          0 Protection           3
    ReadOnly CopyOnWrite

Subsection 3. @ ff863760
   ControlArea: ff8636e8  Starting Sector 3D2 Number Of Sectors 7
   Base Pte     e1b7476c  Ptes In subsect        1 Unused Ptes          0
   Flags              55  Sector Offset          0 Protection           5
```

```
    ReadOnly CopyOnWrite

Subsection 4. @ ff863780
   ControlArea: ff8636e8  Starting Sector 3D9 Number Of Sectors 21f
   Base Pte     e1b74770  Ptes In subsect       44 Unused Ptes          0
   Flags              15  Sector Offset          0 Protection          1
    ReadOnly CopyOnWrite

Subsection 5. @ ff8637a0
   ControlArea: ff8636e8  Starting Sector 5F8 Number Of Sectors 3a
   Base Pte     e1b74880  Ptes In subsect        8 Unused Ptes          0
   Flags              15  Sector Offset          0 Protection          1
    ReadOnly CopyOnWrite
```

!callback

8/26/2019 • 2 minutes to read • Edit Online

The **!callback** extension displays the callback data related to the trap for the specified thread.

```
!callback Address [Number]
```

Parameters

Address
Specifies the hexadecimal address of the thread. If this is -1 or omitted, the current thread is used.

Number
Specifies the number of the desired callback frame. This frame is noted in the display.

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Kdexts.dll

This extension command can only be used with an x86-based target computer.

Additional Information

For information about system traps, see the Windows Driver Kit (WDK) documentation and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

If the system has not experienced a system trap, this extension will not produce useful data.

!calldata

12/5/2018 • 2 minutes to read • Edit Online

The **!calldata** extension displays performance information in the form of procedure call statistics from the named table.

```
!calldata Table
```

Parameters

Table
Name of the table that collects the call data.

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Kdexts.dll

!can_write_kdump

12/5/2018 • 2 minutes to read • Edit Online

The **!can_write_kdump** extension verifies that there is enough disk space on the target computer to write a kernel dump file of the specified type.

```
!can_write_kdump [-dn] [Options]
```

Parameters

-dn
Specifies that the file system on the target computer is an NTFS file system. If this parameter is omitted, then the amount of disk free space cannot be determined, and a warning will be shown. However, the amount of space required will still be displayed.

Options
The following options are valid:

-t
Specifies that the extension should determine if there is enough space for a minidump.

-s
Specifies that the extension should determine if there is enough space for a summary kernel dump. This is the default value.

-f
Specifies that the extension should determine if there is enough space for a full kernel dump.

DLL

Windows 2000	Kext.dll
Windows XP and later	Kext.dll

Remarks

If no *Option* is specified, then the extension will determine if there is enough space for a summary kernel dump.

In the following example, the file system is not specified:

```
kd> !can_write_kdump
Checking kernel summary dump...
WARNING: Can't predict how many pages will be used, assuming worst-case.
Physical memory: 285560 KB
Page file size: 1572864 KB
NO: Page file too small
```

!cbreg

12/5/2018 • 2 minutes to read • Edit Online

The **!cbreg** extension displays CardBus Socket registers and CardBus Exchangable Card Architecture (ExCA) registers.

```
!cbreg [%%]Address
```

Parameters

%%
Indicates that *Address* is a physical address rather than a virtual address.

Address
Specifies the address of the register to be displayed.

DLL

Windows 2000	Kext.dll Kdextx86.dll
Windows XP and later	Kext.dll

The **!cbreg** extension is only available for an x86-based target computer.

Additional Information

The **!exca** extension can be used to display PCIC ExCA registers by socket number.

!cchelp

8/26/2019 • 2 minutes to read • Edit Online

The **!cchelp** extension displays some brief Help text in the Debugger command window for some of the cache management extensions.

```
!cchelp
```

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Kdexts.dll

Additional Information

For information about cache management, see the Microsoft Windows SDK documentation and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

The **!cchelp** extension displays help for the **!bcb**, **!defwrites**, **!finddata**, and **!scm** cache management extensions. Other cache management extensions include **!openmaps** and **!pcm**.

!chklowmem

6/25/2019 • 2 minutes to read • Edit Online

The **!chklowmem** extension determines whether physical memory pages below 4 GB are filled with the required fill pattern on a computer that was booted with the **/pae** and **/nolowmem** options.

```
!chklowmem
```

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Kdexts.dll

Remarks

This extension is useful when you are verifying that kernel-mode drivers operate properly with physical memory above the 4 GB boundary. Typically, drivers fail by truncating a physical address to 32 bits and then in writing below the 4 GB boundary. The **!chklowmem** extension will detect any writes below the 4 GB boundary.

!cmreslist

12/5/2018 • 2 minutes to read • Edit Online

The **!cmreslist** extension displays the CM_RESOURCE_LIST structure for the specified device object.

```
!cmreslist Address
```

Parameters

Address
Specifies the hexadecimal address of the CM_RESOURCE_LIST structure.

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Kdexts.dll

Additional Information

See Plug and Play Debugging for applications of this extension command. For information about the CM_RESOURCE_LIST structure, see the Windows Driver Kit (WDK) documentation.

!cpuinfo

12/5/2018 • 2 minutes to read • Edit Online

The **!cpuinfo** extension displays detailed information about the target computer's CPU.

Syntax

```
!cpuinfo [Processor]
```

Parameters

Processor
Specifies the processor to be displayed. If this is omitted, all processors are displayed.

DLL

Kdexts.dll

Additional Information

For more information about debugging multiprocessor computers, see Multiprocessor Syntax.

Remarks

The **!cpuinfo** extension command can be used when performing local kernel debugging.

Here is an example generated by an x86-based processor:

```
kd> !cpuinfo
CP F/M/S Manufacturer  MHz Update Signature Features
 0 6,1,9 GenuineIntel  198 000000d200000000 000000ff
```

Here is an example generated by an Itanium-based processor:

```
kd> !cpuinfo
CP M/R/F/A Manufacturer     SerialNumber     Features         Speed
 0 2,1,31,0 GenuineIntel     0000000000000000 0000000000000001 1300 Mhz
 1 2,1,31,0 GenuineIntel     0000000000000000 0000000000000001 1300 Mhz
```

The **CP** column indicates the processor number. The **Manufacturer** column specifies the processor manufacturer. The **MHz** or **Speed** column specifies the speed, in MHz, of the processor, if it is available.

For an x86-based processor or an x64-based processor, the **F** column displays the processor family number, the **M** column displays the processor model number, and the **S** column displays the stepping size.

For an Itanium-based processor, the **M** column displays the processor model number, the **R** column displays the processor revision number, the **F** column displays the processor family number, and the **A** column displays the architecture revision number.

Other columns will also appear, depending on your machine's specific architecture.

For details on how to interpret specific values for each entry, as well as any additional columns, consult the processor manual.

!db, !dc, !dd, !dp, !dq, !du, !dw

12/5/2018 • 3 minutes to read • Edit Online

The **!db**, **!dc**, **!dd**, **!dp**, **!dq**, **!du**, and **!dw** extensions display data at the specified physical address on the target computer.

These extension commands should not be confused with the **d* (Display Memory)** command, or with the **!ntsdexts.dp** extension command.

```
!db [Caching] [-m] [PhysicalAddress] [L Size]
!dc [Caching] [-m] [PhysicalAddress] [L Size]
!dd [Caching] [-m] [PhysicalAddress] [L Size]
!dp [Caching] [-m] [PhysicalAddress] [L Size]
!dq [Caching] [-m] [PhysicalAddress] [L Size]
!du [Caching] [-m] [PhysicalAddress] [L Size]
!dw [Caching] [-m] [PhysicalAddress] [L Size]
```

Parameters

Caching
Can be any one of the following values. The *Caching* value must be surrounded by square brackets:

[c]
Causes this extension to read from cached memory.

[uc]
Causes this extension to read from uncached memory.

[wc]
Causes this extension to read from write-combined memory.

-m
Causes memory to be read one unit at a time. For example, **!db -m** reads memory in 8-bit chunks and **!dw -m** reads memory in 16-bit chunks. If your hardware does not support 32-bit physical memory reads, it may be necessary to use the **-m** option. This option does not affect the length or appearance of the display -- it only affects how the memory is accessed.

PhysicalAddress
Specifies the first physical address to be displayed, in hexadecimal format. If this is omitted the first time this command is used, the address defaults to zero. If this is omitted on a subsequent use, the display will begin where the last display ended.

L **** *Size*
Specifies the number of chunks of memory to display. The size of a chunk is determined by the precise extension used.

Environment

Mode	Kernel Mode

DLL

Kext.dll

Additional Information

To write to physical memory, use the **!e*** extensions. For an overview of memory manipulation and a description of other memory-related commands, see Reading and Writing Memory.

Remarks

These extensions each display physical memory, but their display formats and default length differ:

- The **!db** extension displays hexadecimal bytes and their ASCII character equivalents. The default length is 128 bytes.
- The **!dc** extension displays DWORD values and their ASCII character equivalents. The default length is 32 DWORDs (128 total bytes).
- The **!dd** extension displays DWORD values. The default length is 32 DWORDs (128 total bytes).
- The **!dp** extension displays ULONG_PTR values. These are either 32-bit or 64-bit words, depending on the instruction size. The default length is 128 total bytes.
- The **!dq** extension displays ULONG64_PTR values. These are 32-bit words. The default length is 128 total bytes.
- The **!du** extension displays UNICODE characters. The default length is 16 characters (32 total bytes), or until a NULL character is encountered.
- The **!dw** extension displays WORD values. The default length is 64 DWORDs (128 total bytes).

Consequently, using two of these extensions that are distinct with the same value of *Size* will most likely result in a difference in the total amount of memory displayed. For example, using the command **!db L 32** results in 32 bytes being displayed (as hexadecimal bytes), whereas the command **!dd L 32** results in 128 bytes being displayed (as DWORD values).

Here is an example in which the caching attribute flag is needed:

```
kd> !dc e9000
physical memory read at e9000 failed
If you know the caching attributes used for the memory,
try specifying [c], [uc] or [wc], as in !dd [c] <params>.
WARNING: Incorrect use of these flags will cause unpredictable
processor corruption. This may immediately (or at any time in
the future until reboot) result in a system hang, incorrect data
being displayed or other strange crashes and corruption.

kd> !dc [c] e9000
#   e9000 000ea002 000ea002 000ea002 000ea002 ................
#   e9010 000ea002 000ea002 000ea002 000ea002 ................
```

!dbgprint

12/5/2018 • 2 minutes to read • Edit Online

The **!dbgprint** extension displays a string that was previously sent to the **DbgPrint** buffer.

```
!dbgprint
```

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Kdexts.dll

Additional Information

For information about **DbgPrint**, **KdPrint**, **DbgPrintEx**, and **KdPrintEx**, see Sending Output to the Debugger.

Remarks

The kernel-mode routines **DbgPrint**, **KdPrint**, **DbgPrintEx**, and **KdPrintEx** send a formatted string to a buffer on the target computer. The string is automatically displayed in the Debugger Command window on the host computer unless such printing has been disabled.

Generally, messages sent to this buffer are displayed automatically in the Debugger Command window. However, this display can be disabled through the Global Flags (gflags.exe) utility. Moreover, this display does not automatically appear during local kernel debugging. For more information, see The DbgPrint Buffer.

The **!dbgprint** extension causes the contents of this buffer to be displayed (regardless of whether automatic printing has been disabled). It will not show messages that have been filtered out based on their component and importance level. (For details on this filtering, see Reading and Filtering Debugging Messages.)

!dblink

12/5/2018 • 2 minutes to read • Edit Online

The **!dblink** extension displays a linked list in the backward direction.

```
!dblink Address [Count] [Bias]
```

Parameters

Address
Specifies the address of a LIST_ENTRY structure. The display will begin with this node.

Count
Specifies the maximum number of list entries to display. If this is omitted, the default is 32.

Bias
Specifies a mask of bits to ignore in each pointer. Each **Blink** address is ANDed with (NOT *Bias*) before following it to the next location. The default is zero (in other words, do not ignore any bits).

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Kdexts.dll

Remarks

The **!dblink** extension traverses the **Blink** fields of the LIST_ENTRY structure and displays up to four ULONGs at each address. To go in the other direction, use **!dflink**.

The **dl (Display Linked List)** command is more versatile than **!dblink** and **!dflink**.

!dcr

12/5/2018 • 2 minutes to read • Edit Online

The **!dcr** extension displays the default control register (DCR) at the specified address.

```
!dcr Expression [DisplayLevel]
```

Important This command has been deprecated in the Windows Debugger Version 10.0.14257 and later, and is no longer available.

Parameters

Expression
Specifies the hexadecimal address of the DCR to display. The expression **@dcr** can also be used for this parameter. In that case, information about the current processor DCR is displayed.

DisplayLevel
Can be any one of the following options:

0
Causes only the values of each DCR field to be displayed. This is the default value.

1
Causes the display to include more in-depth information about each of the DCR fields that is not reserved or ignored.

2
Causes the display to include more in-depth information about all of the DCR fields, including those that are ignored or reserved.

DLL

Windows 2000	Unavailable
Windows XP and later	Kdexts.dll

This extension command can only be used with an Itanium-based target computer.

Remarks

The DCR specifies default parameters for the processor status register values on interruption. The DCR also specifies some additional global controls, as well as whether or not speculative load faults can be deferred.

Here are a couple of examples:

```
kd> !dcr @dcr
dcr:pp be lc dm dp dk dx dr da dd
1 0 1 1 1 1 1 1 1 1

kd> !dcr @dcr 2

  pp : 1 : Privileged Performance Monitor Default
  be : 0 : Big-Endian Default
  lc : 1 : IA-32 Lock check Enable
  rv : 0 : reserved1
  dm : 1 : Defer TLB Miss faults only
  dp : 1 : Defer Page Not Present faults only
  dk : 1 : Defer Key Miss faults only
  dx : 1 : Defer Key Permission faults only
  dr : 1 : Defer Access Rights faults only
  da : 1 : Defer Access Bit faults only
  dd : 0 : Defer Debug faults only
  rv : 0 : reserved2
```

!dcs

12/5/2018 • 2 minutes to read • Edit Online

The **!dcs** extension is obsolete. To display the PCI configuration space, use **!pci 100***Bus Device Function*.

!deadlock

12/5/2018 • 2 minutes to read • Edit Online

The **!deadlock** extension displays information about deadlocks collected by the **Deadlock Detection** option of Driver Verifier.

```
!deadlock
!deadlock 1
```

DLL

Windows 2000	Unavailable
Windows XP and later	Kdexts.dll

Additional Information

For information about Driver Verifier, see the Windows Driver Kit (WDK) documentation.

Remarks

This extension will only provide useful information if Driver Verifier's **Deadlock Detection** option has detected a lock hierarchy violation and issued **bug check 0xC4** (DRIVER_VERIFIER_DETECTED_VIOLATION).

Without any arguments, the **!deadlock** extension causes the basic lock hierarchy topology to be displayed. If the problem is not a simple cyclical deadlock, this command will describe the situation that has occurred.

The **!deadlock 1** extension causes stack traces to be displayed. The stacks displayed will be the ones active at the time the locks were acquired.

Here is an example:

```
0:kd> !deadlock

Deadlock detected (2 resources in 2 threads):

Thread 0: A B
Thread 1: B A

Where:
Thread 0 = 8d3ba030
Thread 1 = 8d15c030
Lock A =   bba2af30 Type 'Spinlock'
Lock B =   dummy!GlobalLock Type 'Spinlock'
```

This tells you which threads and which locks are involved. However, it is intended to be a summary and may not be enough information to adequately debug the situation.

Use **!deadlock 1** to print out the contents of the call stacks at the time that each lock participating in the deadlock was acquired. Because these are run-time stack traces, they will be more complete if a checked build is being used. On a free build, they may be truncated after as little as one line.

```
0:kd> !deadlock 1

Deadlock detected (2 resources in 2 threads):

Thread 0 (8D14F750) took locks in the following order:

    Lock A -- b7906f30 (Spinlock)
    Stack:   dummy!DummyActivateVcComplete+0x63
             dummy!dummyOpenVcChannels+0x2E1
             dummy!DummyAllocateRecvBufferComplete+0x436
             dummy!DummyAllocateComplete+0x55
             NDIS!ndisMQueuedAllocateSharedHandler+0xC9
             NDIS!ndisWorkerThread+0xEE

    Lock B -- dummy!GlobalLock (Spinlock)
    Stack:   dummy!dummyQueueRecvBuffers+0x2D
             dummy!DummyActivateVcComplete+0x90
             dummy!dummyOpenVcChannels+0x2E1
             dummy!DummyAllocateRecvBufferComplete+0x436
             dummy!DummyAllocateComplete+0x55

Thread 1 (8D903030) took locks in the following order:

    Lock B -- dummy!GlobalLock (Spinlock)
    Stack:   dummy!dummyRxInterruptOnCompletion+0x25D
             dummy!DummyHandleInterrupt+0x32F
             NDIS!ndisMDpcX+0x3C
             ntkrnlpa!KiRetireDpcList+0x5D

    Lock A -- b7906f30 (Spinlock)
    Stack:   << Current stack >>
```

With this information, you have almost everything you need, except the current stack:

```
0: kd> k
ChildEBP RetAddr
f78aae6c 80664c58 ntkrnlpa!DbgBreakPoint
f78aae74 8066523f ntkrnlpa!ViDeadlockReportIssue+0x2f
f78aae9c 806665df ntkrnlpa!ViDeadlockAnalyze+0x253
f78aaee8 8065d944 ntkrnlpa!VfDeadlockAcquireResource+0x20b
f78aaf08 bfd6df46 ntkrnlpa!VerifierKeAcquireSpinLockAtDpcLevel+0x44
f78aafa4 b1bf2d2d dummy!dummyRxInterruptOnCompletion+0x2b5
f78aafc4 bfde9d8c dummy!DummyHandleInterrupt+0x32f
f78aafd8 804b393b NDIS!ndisMDpcX+0x3c
f78aaff4 804b922b ntkrnlpa!KiRetireDpcList+0x5d
```

From this you can see which locks were involved and where they were acquired. This should be enough information for you to debug the deadlock. If the source code is available, you can use the debugger to see exactly where the problem occurred:

```
0: kd> .lines
Line number information will be loaded

0: kd> u dummy!DummyActivateVcComplete+0x63 l1
dummy!DummyActivateVcComplete+63 [d:\nt\drivers\dummy\vc.c @ 2711]:
b1bfe6c9 837d0c00         cmp     dword ptr [ebp+0xc],0x0

0: kd> u dummy!dummyQueueRecvBuffers+0x2D l1
dummy!dummyQueueRecvBuffers+2d [d:\nt\drivers\dummy\receive.c @ 2894]:
b1bf4e39 807d0c01         cmp     byte ptr [ebp+0xc],0x1

0: kd> u dummy!dummyRxInterruptOnCompletion+0x25D l1
dummy!dummyRxInterruptOnCompletion+25d [d:\nt\drivers\dummy\receive.c @ 1424]:
b1bf5d05 85f6             test    esi,esi

0: kd> u dummy!dummyRxInterruptOnCompletion+0x2b5 l1
dummy!dummyRxInterruptOnCompletion+2b5 [d:\nt\drivers\dummy\receive.c @ 1441]:
b1bf5d5d 8b4648           mov     eax,[esi+0x48]
```

Now you know the name of the source file and the line number where the acquisition took place. In this case, the source files will show that the threads behaved as follows:

- Thread 1: **DummyActivateVcComplete** took the **dummy** miniport lock. It then called **dummyQueueRecvBuffers**, which took the **dummy** global lock.
- Thread 2: **dummyRxInterruptOnCompletion** took the global lock. Then, a few lines later, it took the miniport lock.

At this point, the deadlock becomes entirely clear.

!defwrites

12/5/2018 • 2 minutes to read • Edit Online

The **!defwrites** extension displays the values of the kernel variables used by the cache manager.

```
!defwrites
```

DLL

Kdexts.dll

Additional Information

For information about write throttling, see *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

For information about other cache management extensions, use the **!cchelp** extension.

Remarks

When the number of deferred writes ("dirty pages") becomes too large, page writing will be throttled. This extension allows you to see whether your system has reached this point.

Here is an example:

```
kd> !defwrites
*** Cache Write Throttle Analysis ***

        CcTotalDirtyPages:                   0 (       0 Kb)
        CcDirtyPageThreshold:             1538 (    6152 Kb)
        MmAvailablePages:                 2598 (   10392 Kb)
        MmThrottleTop:                     250 (    1000 Kb)
        MmThrottleBottom:                   30 (     120 Kb)
        MmModifiedPageListHead.Total:      699 (    2796 Kb)

Write throttles not engaged
```

In this case, there are no dirty pages. If **CcTotalDirtyPages** reaches 1538 (the value of **CcDirtyPageThreshold**), writing will be delayed until the number of dirty pages is reduced.

!devext

12/5/2018 • 2 minutes to read • Edit Online

The **!devext** extension displays bus-specific device extension information for devices on a variety of buses.

```
!devext Address TypeCode
```

Parameters

Address

Specifies the hexadecimal address of the device extension to be displayed.

TypeCode

Specifies the type of object that owns the device extension to be displayed. Type codes are not case-sensitive. Valid type codes are:

TYPECODE	OBJECT
ISAPNP	ISA PnP device extension
PCMCIA	PCMCIA device extension
HID	HID device extension

DLL

Kdexts.dll

Additional Information

See Plug and Play Debugging for applications of this extension command. For more information about device extensions, see the Windows Driver Kit (WDK) documentation.

Remarks

The **!usbhub**, **!hidfdo**, and **!hidpdo** extensions are obsolete; their functionality has been integrated into **!devext**.

For those object types that are no longer supported by **!devext**, use the **dt (Display Type)** debugger command.

Here is an example for an ISA PnP device extension:

```
kd> !devext e0000165fff32190 ISAPNP
ISA PnP FDO @ 0x00000000, DevExt @ 0xe0000165fff32190, Bus # 196639
Flags (0x854e2530)  DF_ACTIVATED, DF_QUERY_STOPPED,
                    DF_STOPPED, DF_RESTARTED_NOMOVE,
                    DF_BUS
                    Unknown flags 0x054e2000

NumberCSNs           - -536870912
ReadDataPort         - 0x0000000d (mapped)
AddressPort          - 0x00000000 (not mapped)
CommandPort          - 0x00000000 (not mapped)
DeviceList           - 0xe000000085007b50
CardList             - 0x00000000
PhysicalBusDevice    - 0x00000000
AttachedDevice       - 0x00000000
SystemPowerState     - Unspecified
DevicePowerState     - Unspecified
```

Here is an example for a PCI device:

```
kd> !devext e0000000858c31b0 PCI
PDO Extension, Bus 0x0, Device 0, Function 0.
  DevObj 0xe0000000858c3060 PCI Parent Bus FDO DevExt 0xe0000000858c4960
  Device State = PciNotStarted
  Vendor ID 8086 (INTEL)  Device ID 123D
  Class Base/Sub 08/00  (Base System Device/Interrupt Controller)
  Programming Interface: 20, Revision: 01, IntPin: 00, Line Raw/Adj 00/00
  Enables ((cmd & 7) = 106): BM   Capabilities Pointer = <none>
  CurrentState:          System Working,  Device D0
  WakeLevel:             System Unspecified,  Device Unspecified
  Requirements: <none>
```

!devhandles

12/5/2018 • 2 minutes to read • Edit Online

The **!devhandles** extension displays the open handles for the specified device.

```
!devhandles Address
```

Parameters

Address
Specifies the address of the device for which to display the open handles.

DLL

Windows 2000	Unavailable
Windows XP and later	Kdexts.dll

Remarks

To display complete handle information, this extension requires private symbols.

The address of a device object can be obtained using the **!drvobj** or **!devnode** extensions.

Here is a truncated example:

```
lkd> !devhandles 0x841153d8

Checking handle table for process 0x840d3940
Handle table at 95fea000 with 578 Entries in use

Checking handle table for process 0x86951d90
Handle table at 8a8ef000 with 28 Entries in use

...

Checking handle table for process 0x87e63650
Handle table at 947bc000 with 308 Entries in use

Checking handle table for process 0x87e6f4f0
00000000: Unable to read handle table
```

!devnode

8/26/2019 • 2 minutes to read • Edit Online

The **!devnode** extension displays information about a node in the device tree.

```
!devnode Address [Flags] [Service]
!devnode 1
!devnode 2
```

Parameters

Address
Specifies the hexadecimal address of the device extension whose node is to be displayed. If this is zero, the root of the main device tree is displayed.

Flags
Specifies the level of output to be displayed. This can be any combination of the following bits:

Bit 0 (0x1)
Causes the display to include all children of the device node.

Bit 1 (0x2)
Causes the display to include resources used (CM_RESOURCE_LIST). These include the boot configuration reported by IRP_MN_QUERY_RESOURCES, as well as the resources allocated to the device in the *AllocatedResources* parameter of IRP_MN_START_DEVICE.

Bit 2 (0x4)
Causes the display to include resources required (IO_RESOURCE_REQUIREMENTS_LIST) as reported by IRP_MN_FILTER_RESOURCE_REQUIREMENTS.

Bit 3 (0x8)
Causes the display to include a list of translated resources as allocated to the device in the *AllocatedResourcesTranslated* parameter of IRP_MN_START_DEVICE.

Bit 4 (0x10)
Specifies that only device nodes that are not started should be displayed.

Bit 5 (0x20)
Specifies that only device nodes with problems should be displayed. (These are nodes that contain the flag bits DNF_HAS_PROBLEM or DNF_HAS_PRIVATE_PROBLEM.)

Service
Specifies the name of a service. If this is included, only those device nodes driven by this service will be displayed. (If *Flags* includes bit 0x1, device nodes driven by this service and all their children will be displayed.)

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Kdexts.dll

Additional Information

See Plug and Play Debugging for applications of this extension command. For information about device trees, see the Windows Driver Kit (WDK) documentation and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

The **!devnode 1** command lists all pending removals of device objects.

The **!devnode 2** command lists all pending ejects of device objects.

You can use **!devnode 0 1** to see the entire device tree.

!devobj

12/5/2018 • 2 minutes to read • Edit Online

The **!devobj** extension displays detailed information about a DEVICE_OBJECT structure.

```
!devobj DeviceObject
```

Parameters

DeviceObject
Specifies the device object. This can be the hexadecimal address of this structure or the name of the device.

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Kdexts.dll

Additional Information

See Plug and Play Debugging for examples and applications of this extension command. For information about device objects, see the Windows Driver Kit (WDK) documentation and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

If *DeviceObject* specifies the name of the device but supplies no prefix, the prefix "\Device\" is assumed. Note that this command will check to see if *DeviceObject* is a valid address or device name before using the expression evaluator.

The information displayed includes the device name of the object, information about the device's current IRP, and a list of addresses of any pending IRPs in the device's queue. It also includes information about device objects layered on top of this object (listed as "AttachedDevice") and those layered under this object (listed as "AttachedTo").

The address of a device object can be obtained using the **!drvobj** or **!devnode** extensions.

Here is one example:

```
kd> !devnode
Dumping IopRootDeviceNode (= 0x80e203b8)
DevNode 0x80e203b8 for PDO 0x80e204f8
 Parent 0000000000   Sibling 0000000000   Child 0x80e56dc8
  InstancePath is "HTREE\ROOT\0"
  State = DeviceNodeStarted (0x308)
  Previous State = DeviceNodeEnumerateCompletion (0x30d)
  StateHistory[04] = DeviceNodeEnumerateCompletion (0x30d)
  StateHistory[03] = DeviceNodeStarted (0x308)
  StateHistory[02] = DeviceNodeEnumerateCompletion (0x30d)
  StateHistory[01] = DeviceNodeStarted (0x308)
  StateHistory[00] = DeviceNodeUninitialized (0x301)
  StateHistory[19] = Unknown State (0x0)
  .....
  StateHistory[05] = Unknown State (0x0)
  Flags (0x00000131)  DNF_MADEUP, DNF_ENUMERATED,
                      DNF_IDS_QUERIED, DNF_NO_RESOURCE_REQUIRED
  DisableableDepends = 11 (from children)

kd> !devobj 80e204f8
Device object (80e204f8) is for:
  \Driver\PnpManager DriverObject 80e20610
Current Irp 00000000 RefCount 0 Type 00000004 Flags 00001000
DevExt 80e205b0 DevObjExt 80e205b8 DevNode 80e203b8
ExtensionFlags (0000000000)
Device queue is not busy.
```

!devstack

12/5/2018 • 2 minutes to read • Edit Online

The **!devstack** extension displays a formatted view of the device stack associated with a device object.

```
!devstack DeviceObject
```

Parameters

DeviceObject
Specifies the device object. This can be the hexadecimal address of the DEVICE_OBJECT structure or the name of the device.

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Kdexts.dll

Additional Information

For information about device stacks, see the Windows Driver Kit (WDK) documentation.

Remarks

If *DeviceObject* specifies the name of the device but supplies no prefix, the prefix "\Device\" is assumed. Note that this command will check to see if *DeviceObject* is a valid address or device name before using the expression evaluator.

Here is an example:

```
kd> !devstack e000000085007b50
 !DevObj   !DrvObj            !DevExt   ObjectName
  e0000165fff32040  \Driver\kmixer     e0000165fff32190
> e000000085007b50  \Driver\swenum     e000000085007ca0  KSENUM#00000005
!DevNode e0000165fff2e010 :
  DeviceInst is "SW\{b7eafdc0-a680-11d0-96d8-00aa0051e51d}\{9B365890-165F-11D0-A195-0020AFD156E4}"
 ServiceName is "kmixer"
```

!dflink

12/5/2018 • 2 minutes to read • Edit Online

The **!dflink** extension displays a linked list in the forward direction.

```
!dflink Address [Count] [Bias]
```

Parameters

Address
Specifies the address of a LIST_ENTRY structure. The display will begin with this node.

Count
Specifies the maximum number of list entries to display. If this is omitted, the default is 32.

Bias
Specifies a mask of bits to ignore in each pointer. Each **Flink** address is ANDed with (NOT *Bias*) before following it to the next location. The default is zero (in other words, do not ignore any bits).

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Kdexts.dll

Remarks

The **!dflink** extension traverses the **Flink** fields of the LIST_ENTRY structure and displays up to four ULONGs at each address. To go in the other direction, use **!dblink**.

The **dl (Display Linked List)** command is more versatile than **!dblink** and **!dflink**.

!diskspace

12/5/2018 • 2 minutes to read • Edit Online

The **!diskspace** extension displays the amount of free space on a hard disk of the target computer.

```
!diskspace Drive[:]
```

Parameters

Drive
Specifies the drive letter of the disk. The colon (:) after *Drive* is optional.

DLL

Windows 2000	Kext.dll
Windows XP and later	Kext.dll

Remarks

Here is an example:

```
kd> !diskspace c:
Checking Free Space for c: ..........
   Cluster Size 0 KB
 Total Clusters 4192901 KB
  Free Clusters 1350795 KB
    Total Space 1 GB (2096450 KB)
     Free Space 659.567871 MB (675397 KB)

kd> !diskspace f:
Checking Free Space for f:
f: is a CDROM drive. This function is not supported!
```

!dma

8/26/2019 • 3 minutes to read • Edit Online

The **!dma** extension displays information about the Direct Memory Access (DMA) subsystem, and the **DMA Verifier** option of Driver Verifier.

```
!dma
!dma Adapter [Flags]
```

Parameters

Adapter
Specifies the hexadecimal address of the DMA adapter to be displayed. If this is zero, all DMA adapters will be displayed.

Flags
Specifies the information to include in the display. This can be any combination of the following bits. The default is 0x1.

Bit 0 (0x1)
Causes the display to include generic adapter information.

Bit 1 (0x2)
Causes the display to include map register information. (Only when DMA Verification is active.)

Bit 2 (0x4)
Causes the display to include common buffer information. (Only when DMA Verification is active.)

Bit 3 (0x8)
Causes the display to include scatter/gather list information. (Only when DMA Verification is active.)

Bit 4 (0x10)
Causes the display to include the device description for the hardware device. (Only when DMA Verification is active.)

Bit 5 (0x20)
Causes the display to include Wait context block information.

DLL

Windows 2000	Unavailable
Windows XP and later	Kdexts.dll

Additional Information

For information about Driver Verifier, see the Windows Driver Kit (WDK) documentation. For information about DMA, see the Windows Driver Kit (WDK) documentation and *Microsoft Windows Internals* by Mark Russinovich David Solomon.

Remarks

Invalid arguments (for example, **!dma 1**) generate a brief help text.

When the **!dma** extension is used with no parameters, it displays a concise list of all DMA adapters and their addresses. This can be used to obtain the address of an adapter for use in the longer versions of this command.

Here is an example of how this extension can be used when the Driver Verifier's **DMA Verification** option is active:

```
0:kd> !dma

Dumping all DMA adapters...

Adapter: 82faebd0     Owner: SCSIPORT!ScsiPortGetUncachedExtension
Adapter: 82f88930     Owner: SCSIPORT!ScsiPortGetUncachedExtension
Adapter: 82f06cd0     Owner: NDIS!NdisMAllocateMapRegisters
Master adapter: 80076800
```

From this output, you can see that there are three DMA adapters in the system. SCSIPORT owns two and NDIS owns the third. To examine the NDIS adapter in detail, use the **!dma** extension with its address:

```
0:kd> !dma  82f06cd0
Adapter: 82f06cd0     Owner: NDIS!NdisMAllocateMapRegisters (0x9fe24351)
 MasterAdapter:       00000000
   Adapter base Va      00000000
   Map register base:   00000000
 WCB:                 82f2b604
   Map registers: 00000000 mapped, 00000000 allocated, 00000002 max

   Dma verifier additional information:
   DeviceObject: 82f98690
   Map registers:        00000840 allocated, 00000000 freed
   Scatter-gather lists: 00000000 allocated, 00000000 freed
   Common buffers:       00000004 allocated, 00000000 freed
   Adapter channels:     00000420 allocated, 00000420 freed
   Bytes mapped since last flush: 000000f2
```

The first block of data is specific information that a HAL developer can use to debug the problem. For your purposes, the data below "Dma verifier additional information" is what is interesting. In this example, you see that NDIS has allocated 0x840 map registers. This is a huge number, especially because NDIS had indicated that it planned to use a maximum of two map registers. This adapter apparently does not use scatter/gather lists and has put away all its adapter channels. Look at the map registers in more detail:

```
0:kd> !dma  82f06cd0 2
Adapter: 82f06cd0     Owner: NDIS!NdisMAllocateMapRegisters
...

  Map register file 82f06c58 (0/2 mapped)
     Double buffer mdl: 82f2c188
     Map registers:
        82f06c80: Not mapped
        82f06c8c: Not mapped

  Map register file 82f06228 (1/2 mapped)
     Double buffer mdl: 82f1b678
     Map registers:
        82f06250: 00bc bytes mapped to f83c003c
        82f0625c: Not mapped

  Map register file 82fa5ad8 (1/2 mapped)
     Double buffer mdl: 82f1b048
     Map registers:
        82fa5b00: 0036 bytes mapped to 82d17102
        82fa5b0c: Not mapped
...
```

In this example, you see that certain map registers have been mapped. Each *map register file* is an allocation of map registers by the driver. In other words, it represents a single call to **AllocateAdapterChannel**. NDIS collects a large number of these map register files, while some drivers create them one at a time and dispose of them when they are finished.

The map register files are also the addresses that are returned to the device under the name "MapRegisterBase". Note that DMA verifier only hooks the first 64 map registers for each driver. The rest are ignored for reasons of space (each map register represents three physical pages).

In this example, two map register files are in use. This means that the driver has mapped a buffer so it is visible to the hardware. In the first case, 0xBC bytes are mapped to the system virtual address 0xF83C003C.

An examination of the common buffers reveals:

```
0:kd> !dma  82f06cd0 4
Adapter: 82f06cd0     Owner: NDIS!NdisMAllocateMapRegisters
...
   Common buffer allocated by NDIS!NdisMAllocateSharedMemory:
      Length:           1000
      Virtual address:  82e77000
      Physical address: 2a77000

   Common buffer allocated by NDIS!NdisMAllocateSharedMemory:
      Length:           12010
      Virtual address:  82e817f8
      Physical address: 2a817f8

   Common buffer allocated by NDIS!NdisMAllocateSharedMemory:
      Length:           4300
      Virtual address:  82e95680
      Physical address: 2a95680

   Common buffer allocated by NDIS!NdisMAllocateSharedMemory:
      Length:           4800
      Virtual address:  82e9d400
      Physical address: 2a9d400
```

This is fairly straightforward; there are four common buffers of varying lengths. The physical and virtual addresses are all given.

!dpa

12/5/2018 • 2 minutes to read • Edit Online

The **!dpa** extension displays pool allocation information.

```
!dpa Options
!dpa -?
```

Parameters

Options
Must be exactly one of the following options:

-c
Displays current pool allocation statistics.

-v
Displays all current pool allocations.

-vs
Causes the display to include stack traces.

-f
Displays failed pool allocations.

-r
Displays free pool allocations.

-p **** *Ptr*
Displays all pool allocations that contain the pointer *Ptr*.

-?
Displays some brief Help text for this extension in the Debugger Command window.

DLL

Windows 2000	Unavailable
Windows XP and later	Kdexts.dll

Remarks

Pool instrumentation must be enabled in Win32k.sys in order for this extension to work.

!dpcs

8/26/2019 • 2 minutes to read • Edit Online

The **!dpcs** extension displays the deferred procedure call (DPC) queues for a specified processor.

```
!dpcs [Processor]
```

Parameters

Processor
Specifies a processor. If *Processor* is omitted, then the DPC queues for all processors are displayed.

DLL

Windows 2000	Unavailable
Windows XP	Unavailable
Windows Server 2003 and later	Kdexts.dll

Additional Information

For information about DPCs, see the Windows Driver Kit (WDK) documentation and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

!driveinfo

12/5/2018 • 2 minutes to read • Edit Online

The **!driveinfo** extension displays volume information for the specified drive.

```
!driveinfo Drive[:]
!driveinfo
```dbgcmd

## <span id="Parameters"></span><span id="parameters"></span><span id="PARAMETERS"></span>Parameters


<span id="_____Drive_____"></span><span id="_____drive_____"></span><span id="_____DRIVE_____"></span>
*Drive*
Specifies a drive. The colon at the end of the drive name is optional.

<span id="_____No_parameters_____"></span><span id="_____no_parameters_____"></span><span
id="_____NO_PARAMETERS_____"></span> *No parameters*
Displays some brief Help text for this extension in the Debugger Command window.

### <span id="DLL"></span><span id="dll"></span>DLL

<table>
<colgroup>
<col width="50%" />
<col width="50%" />
</colgroup>
<tbody>
<tr class="odd">
<td align="left"><p><strong>Windows 2000</strong></p></td>
<td align="left"><p>Unavailable</p></td>
</tr>
<tr class="even">
<td align="left"><p><strong>Windows XP and later</strong></p></td>
<td align="left"><p>Kdexts.dll</p></td>
</tr>
</tbody>
</table>



Remarks
-------

The drive information displayed by this extension is obtained by querying the underlying file system; for
example:

```dbgcmd
kd> !driveinfo c:
Drive c:, DriveObject e136cd88
    Directory Object: e1001408  Name: C:
        Target String is '\Device\HarddiskVolume1'
        Drive Letter Index is 3 (C:)
 Volume DevObj: 82254a68
    Vpb: 822549e0  DeviceObject: 82270718
 FileSystem: \FileSystem\Ntfs
    Volume has 0x229236 (free) / 0x2ee1a7 (total) clusters of size 0x1000
    8850.21 of 12001.7 MB free
```

!drivers

8/27/2019 • 2 minutes to read • Edit Online

> **NOTE**
> In Windows XP and later versions of Windows, the **!drivers** extension is obsolete. To display information about loaded drivers and other modules, use the **lm** command.

The command lm t n displays information in a format very similar to the old **!drivers** extension. However, this command will not display the memory usage of the drivers as the **!drivers** extension did. It will only display the drivers' start and end addresses, image names, and timestamps. The **!vm** and **!memusage** extensions can be used to display memory usage statistics.

```
!drivers [Flags]
```

Parameters

Flags
Can be any combination of the following values. (The default is 0x0.)

Bit 0 (0x1)
Causes the display to include information about resident and standby memory.

Bit 1 (0x2)
If this bit is set and bit 2 (0x4) is not set, the display will include information about resident, standby, and locked memory, as well as the loader entry address. If bit 2 is set, this causes the display to be a much longer and more detailed list of the driver image. Information about the headers is included, as is section information.

Bit 2 (0x4)
Causes the display to be a much longer and more detailed list of the driver image. Information about each section is included. If bit 1 (0x2) is set, this will also include header information.

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Unavailable

Additional Information

See Plug and Play Debugging for applications of this extension command. For information about drivers and their memory use, see the Windows Driver Kit (WDK) documentation and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

An explanation of this command's display is given in the following table:

COLUMN	MEANING
Base	The starting address of the device driver code, in hexadecimal. When the memory address used by the code that causes a stop falls between the base address for a driver and the base address for the next driver in the list, that driver is frequently the cause of the fault. For instance, the base for Ncrc810.sys is 0x80654000. Any address between that and 0x8065a000 belongs to this driver.
Code Size	The size, in kilobytes, of the driver code, in both hexadecimal and decimal.
Data Size	The amount of space, in kilobytes, allocated to the driver for data, in both hexadecimal and decimal.
Locked	(Only when Flag 0x2 is used) The amount of memory locked by the driver.
Resident	(Only when Flag 0x1 or 0x2 is used) The amount of the driver's memory that actually resides in physical memory.
Standby	(Only when Flag 0x1 or 0x2 is used) The amount of the driver's memory that is on standby.
Loader Entry	(Only when Flag 0x2 is used) The loader entry address.
Driver Name	The driver file name.
Creation Time	The link date of the driver. Do not confuse this with the file date of the driver, which can be set by external tools. The link date is set by the compiler when a driver or executable file is compiled. It should be close to the file date, but it is not always the same.

The following is a truncated example of this command:

```
kd> !drivers
Loaded System Driver Summary
Base     Code Size      Data Size      Driver Name  Creation Time
80080000 f76c0 (989 kb) 1f100 (124 kb) ntoskrnl.exe Fri May 26 15:13:00
80400000 d980  ( 54 kb) 4040  ( 16 kb) hal.dll      Tue May 16 16:50:34
80654000 3f00  ( 15 kb) 1060   ( 4 kb) ncrc810.sys  Fri May 05 20:07:04
8065a000 a460  ( 41 kb) 1e80   ( 7 kb) SCSIPORT.SYS Fri May 05 20:08:05
```

!drvobj

12/5/2018 • 2 minutes to read • Edit Online

The **!drvobj** extension displays detailed information about a DRIVER_OBJECT.

```
!drvobj DriverObject [Flags]
```

Parameters

DriverObject
Specifies the driver object. This can be the hexadecimal address of the DRIVER_OBJECT structure or the name of the driver.

Flags
Can be any combination of the following bits. (The default is 0x01.)

Bit 0 (0x1)
Causes the display to include device objects owned by the driver.

Bit 1 (0x2)
Causes the display to include entry points for the driver's dispatch routines.

Bit 2 (0x4)
Lists with detailed information the device objects owned by the driver (requires bit 0 (0x1)).

DLL

Kdexts.dll

Additional Information

See Plug and Play Debugging for examples and applications of this extension command. For information about driver objects, see the Windows Driver Kit (WDK) documentation and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

If *DriverObject* specifies the name of the device but supplies no prefix, the prefix "\Driver\" is assumed. Note that this command will check to see if *DriverObject* is a valid address or device name before using the expression evaluator.

If *DriverObject* is an address, it must be the address of the DRIVER_OBJECT structure. This can be obtained by examining the arguments passed to the driver's **DriverEntry** routine.

This extension command will display a list of all device objects created by a specified driver. It will also display all fast I/O routines registered with this driver object.

The following is an example for the Symbios Logic 810 SCSI miniport driver:

```
kd> bp DriverEntry          //  breakpoint at DriverEntry

kd> g
symc810!DriverEntry+0x40:
80006a20: b07e0050 stl     t2,50(sp)

kd> r a0  //address of DevObj (the first parameter)
a0=809d5550

kd> !drvobj 809d5550   //  display the driver object
Driver object is for:
\Driver\symc810
Device Object list:
809d50d0
```

You can also use **!devobj 809d50d0** to get information about the device object.

!dskheap

8/26/2019 • 2 minutes to read • Edit Online

The **!dskheap** extension displays desktop heap information for a specified session.

```
!dskheap [-v] [-s SessionID]
```

Parameters

-v
Causes the display to include more detailed output.

-s **** *SessionID*
Specifies a session. If this parameter is omitted, then the desktop heap information for session 0 is displayed.

DLL

Windows 2000	Unavailable
Windows XP and later	Kdexts.dll

Additional Information

For information about desktops or desktop heaps, see the Microsoft Windows SDK documentation and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

The desktop heap information for the session is arranged by window station.

Here are a couple of examples:

```
kd> !dskheap -s 3
##   Winstation\Desktop            Heap Size(KB)   Used Rate(%)

  WinSta0\Screen-saver              3072                0%
  WinSta0\Default                   3072                0%
  WinSta0\Disconnect                  64                4%
##   WinSta0\Winlogon                  128                5%

                Total Desktop: (    6336 KB -   4 desktops)
#                 Session ID:  3

kd> !dskheap
##   Winstation\Desktop            Heap Size(KB)   Used Rate(%)

  WinSta0\Default                   3072                0%
  WinSta0\Disconnect                  64                4%
  WinSta0\Winlogon                  128                 9%
  Service-0x0-3e7$\Default          512                 4%
  Service-0x0-3e5$\Default          512                 0%
  Service-0x0-3e4$\Default          512                 1%
##   SAWinSta\SADesktop                512                0%

                Total Desktop: (    5312 KB -   7 desktops)
#                 Session ID:  0
```

!eb, !ed

12/5/2018 • 2 minutes to read • Edit Online

The **!eb** and **!ed** extensions write a sequence of values into a specified physical address.

These extension commands should not be confused with the **e* (Enter Values)** command.

```
!eb [Flag] PhysicalAddress Data [ ... ]
!ed [Flag] PhysicalAddress Data [ ... ]
```

Parameters

Flag
Can be any one of the following values. The *Flag* value must be surrounded by square brackets:

[c]
Writes to cached memory.

[uc]
Writes to uncached memory.

[wc]
Writes to write-combined memory.

PhysicalAddress
Specifies the first physical address on the target computer to which the data will be written, in hexadecimal.

Data
Specifies one or more values to be written sequentially into physical memory. Enter these values in hexadecimal format. For the **!eb** extension, each value must be 1 byte (two hexadecimal digits). For the **!ed** extension, each value must be one DWORD (eight hexadecimal digits). You can include any number of *Data* values on one line. To separate multiple values, use commas or spaces.

DLL

Windows 2000	Kext.dll Kdextx86.dll
Windows XP and later	Kext.dll

Additional Information

To read physical memory, use the **!d*** extensions. For an overview of memory manipulation and a description of other memory-related commands, see Reading and Writing Memory.

!ecb, !ecd, !ecw

12/5/2018 • 2 minutes to read • Edit Online

The **!ecb**, **!ecd**, and **!ecw** extensions write to the PCI configuration space.

```
!ec Bus.Device.Function Offset Data
```

Parameters

Bus
Specifies the bus. *Bus* can range from 0 to 0xFF.

Device
Specifies the slot device number for the device.

Function
Specifies the slot function number for the device.

Offset
Specifies the address at which to write.

Data
Specifies the value to be written. For the **!ecb** extension, *Data* must be 1 byte (two hexadecimal digits). For the **!ecw** extension, *Data* must be one WORD (four hexadecimal digits). For the **!ecd** extension, *Data* must be one DWORD (eight hexadecimal digits).

DLL

Windows 2000	Kext.dll
Windows XP and later	Kext.dll

These extension commands can only be used with an x86-based target computer.

Additional Information

See Plug and Play Debugging for applications of this extension command, and some additional examples. For information about PCI buses, see the Windows Driver Kit (WDK) documentation.

Remarks

You cannot use these extension commands to write a sequence of *Data* values. This can only be done by repeated use of this extension.

To display the PCI configuration space, use **!pci 100***Bus Device Function*.

!ecs

12/5/2018 • 2 minutes to read • Edit Online

The **!ecs** extension is obsolete. To edit the PCI configuration space, use **!ecb**, **!ecd**, or **!ecw**.

!errlog

10/23/2019 • 2 minutes to read • Edit Online

The **!errlog** extension displays the contents of any pending entries in the I/O system's error log.

```
!errlog
```

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Kdexts.dll

Additional Information

For information about **IoWriteErrorLogEntry**, see the Windows Driver Kit (WDK) documentation.

Remarks

This command displays information about any pending events in the I/O system's error log. These are events queued by calls to the **IoWriteErrorLogEntry** function, to be written to the system's event log for subsequent viewing by the **Event Viewer**.

Only entries that were queued by **IoWriteErrorLogEntry** but have not been committed to the error log will be displayed.

This command can be used as a diagnostic aid after a system crash because it reveals pending error information that was unable to be committed to the error log before the system halted.

!errpkt

12/5/2018 • 2 minutes to read • Edit Online

The **!errpkt** extension displays the contents of a Windows Hardware Error Architecture (WHEA) hardware error packet.

```
!errpkt Address
```

Parameters

Address
Specifies the address of the hardware error packet.

DLL

Windows 2000	Unavailable
Windows XP	Unavailable
Windows Server 2003	Unavailable
Windows Vista and later	Kdexts.dll

This extension can be used only in Windows Vista and later versions of Windows.

Additional Information

The **!whea** and **!errrec** extensions can be used to display additional WHEA information. For general information about WHEA, see Windows Hardware Error Architecture (WHEA) in the Windows Driver Kit (WDK) documentation.

Remarks

The following example shows the output of the **!errpkt** extension:

```
3: kd> !errpkt fffffa8007cf44da
   WHEA Error Packet Info Section (@ fffffa8007cf44da)
   Flags            : 0x00000000
   Size             : 0x218
   RawDataLength    : 0x392
   Context          : 0x0000000000000000
   ErrorType        : 0x0 - Processor
   ErrorSeverity    : 0x1 - Fatal
   ErrorSourceId    : 0x0
   ErrorSourceType  : 0x0 - MCE
   Version          : 00000002
   Cpu              : 0000000000000002
   RawDataFormat    : 0x2 - Intel64 MCA

   Raw Data         : Located @ FFFFFA8007CF45F2

Processor Error: (Internal processor error)
This error means either the processor is damaged or perhaps
voltage and/or temperature thresholds have been exceeded.
If the problem continues to occur, replace the processor.
Processor Number : 2
Bank Number      : 0
   Status  :               0
   Address : 0000000000000000 (I)
   Misc    : 0000000000000000 (I)
```

!errrec

12/5/2018 • 2 minutes to read • Edit Online

The **!errrec** extension displays the contents of a Windows Hardware Error Architecture (WHEA) error record.

```
!errrec Address
```

Parameters

Address
Specifies the address of the error record.

DLL

Windows 2000	Unavailable
Windows XP	Unavailable
Windows Server 2003	Unavailable
Windows Vista and later	Kdexts.dll

This extension can be used only in Windows Vista and later versions of Windows.

Additional Information

The **!whea** and **!errpkt** extensions can be used to display additional WHEA information. For general information about WHEA, see Windows Hardware Error Architecture (WHEA) in the Windows Driver Kit (WDK) documentation.

Remarks

The following example shows how the **!whea** extension can be used to obtain the address of an error record, and then the contents of this record can be displayed by the **!errrec** extension:

```
3: kd> !whea
Error Source Table @ fffff800019ca250
13 Error Sources
Error Source 0 @ fffffa80064132c0
   Notify Type      : Machine Check Exception
   Type             : 0x0 (MCE)
   Error Count      : 8
   Record Count     : 10
   Record Length    : c3e
   Error Records    : wrapper @ fffffa8007cf4000  record @ fffffa8007cf4030

. . . .

# 3: kd> !errrec fffffa8007cf4030
===============================================================================
## Common Platform Error Record @ fffffa8007cf4030
```

```
--------------------------------------------------------------------------------
Revision      : 2.1
Record Id     : 01c9c7ff04e0ff7d
Severity      : Fatal (1)
Length        : 1730
Creator       : Microsoft
Notify Type   : Machine Check Exception
Timestamp     : 4/28/2009 12:54:47
Flags         : 0x00000000
#
================================================================================
## Section 0     : Processor Generic
--------------------------------------------------------------------------------
Descriptor    @ fffffa8007cf40b0
Section       @ fffffa8007cf4188
Offset        : 344
Length        : 192
Flags         : 0x00000001 Primary
Severity      : Fatal

No valid data fields are present.
#
================================================================================
## Section 1     : {390f56d5-ca86-4649-95c4-73a408ae5834}
--------------------------------------------------------------------------------
Descriptor    @ fffffa8007cf40f8
Section       @ fffffa8007cf4248
Offset        : 536
Length        : 658
Flags         : 0x00000000
Severity      : Fatal

*** Unknown section format ***
#
================================================================================
## Section 2     : Error Packet
--------------------------------------------------------------------------------
Descriptor    @ fffffa8007cf4140
Section       @ fffffa8007cf44da
Offset        : 1194
Length        : 536
Flags         : 0x00000000
Severity      : Fatal

   WHEA Error Packet Info Section (@ fffffa8007cf44da)
   Flags            : 0x00000000
   Size             : 0x218
   RawDataLength    : 0x392
   Context          : 0x0000000000000000
   ErrorType        : 0x0 - Processor
   ErrorSeverity    : 0x1 - Fatal
   ErrorSourceId    : 0x0
   ErrorSourceType  : 0x0 - MCE
   Version          : 00000002
   Cpu              : 0000000000000002
   RawDataFormat    : 0x2 - Intel64 MCA

   Raw Data         : Located @ FFFFFA8007CF45F2

Processor Error: (Internal processor error)
This error means either the processor is damaged or perhaps
voltage and/or temperature thresholds have been exceeded.
If the problem continues to occur, replace the processor.
Processor Number : 2
Bank Number      : 0
   Status  :               0
   Address : 0000000000000000 (I)
   Misc    : 0000000000000000 (I)
```

!exca

12/5/2018 • 2 minutes to read • Edit Online

The **!exca** extension displays PC-Card Interrupt Controller (PCIC) Exchangable Card Architecture (ExCA) registers.

```
!exca BasePort.SocketNumber
```

Parameters

BasePort
Specifies the base port of the PCIC.

SocketNumber
Specifies the socket number of the ExCA register on the PCIC.

DLL

Windows 2000	Kext.dll Kdextx86.dll
Windows XP and later	Kext.dll

The **!exca** extension is only available for an x86-based target computer.

Additional Information

The **!cbreg** extension can be used to display CardBus Socket registers and CardBus ExCA registers by address.

!filecache

8/26/2019 • 3 minutes to read • Edit Online

The **!filecache** extension displays information regarding the system file cache memory and PTE use.

```
!filecache [Flags]
```

Parameters

Flags
Optional. Default value is 0x0. Set *Flags* to 0x1 to sort the output by shared cache map. This way you can locate all the system cache views for a given control area.

DLL

Kdexts.dll

Additional Information

For information about file system drivers, see the Windows Driver Kit (WDK) documentation and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

Each line of this extension's output represents a virtual address control block (VACB). When named files are mapped into the VACB, the names of these files are displayed. If "no name for file" is specified, this means that this VACB is being used to cache metadata.

Here is an example of the output from this extension from a Windows XP system:

```
kd> !filecache
***** Dump file cache******
  Reading and sorting VACBs ...
  Removed 1811 nonactive VACBs, processing 235 active VACBs ...
File Cache Information
  Current size 28256 kb
  Peak size    30624 kb
  235 Control Areas
Skipping view @ c1040000 - no VACB, but PTE is valid!
  Loading file cache database (100% of 131072 PTEs)
  SkippedPageTableReads = 44
  File cache has 4056 valid pages


  Usage Summary (in Kb):
Control Valid Standby/Dirty Shared Locked Name
817da668     4      0     0     0  $MftMirr
8177ae68   304    920     0     0  $LogFile
81776160   188      0     0     0  $BitMap
817cf370     4      0     0     0  $Mft
81776a00     8      0     0     0  $Directory
817cfdd0     4      0     0     0  $Directory
81776740    36      0     0     0    No Name for File
817cf7c8    20      0     0     0  $Directory
817cfb98   304      0     0     0  $Directory
8177be40    16      0     0     0  $Directory
817dadc0  2128     68     0     0  $Mft
```

```
817cf008     4      0     0     0  $Directory
817d0258     8      4     0     0  $Directory
817763f8     4      0     0     0  $Directory
...
8173f058     4      0     0     0  $Directory
8173e628    32      0     0     0  $Directory
8173e4c8    32      0     0     0  $Directory
8173da38     4      0     0     0  $Directory
817761f8     4      0     0     0  $Directory
81740530    32      0     0     0  $Directory
8173d518     4      0     0     0  $Directory
817d9560     8      0     0     0  $Directory
8173f868     4      0     0     0  $Directory
8173fc00     4      0     0     0  $Directory
81737278     4      0     0     0  $MftMirr
81737c88    44      0     0     0  $LogFile
81735fa0    48      0     0     0  $Mft
81737e88   188      0     0     0  $BitMap
817380b0     4      0     0     0  $Mft
817399e0     4      0     0     0  $Directory
817382b8     4      0     0     0  $Directory
817388d8    12      0     0     0    No Name for File
81735500     8      0     0     0  $Directory
81718e38   232      0     0     0  default
81735d40    48     20     0     0  SECURITY
81723008  8632      0     0     0  software
816da3a0    24     44     0     0  SAM
8173dfa0     4      0     0     0  $Directory
...
8173ba90     4      0     0     0  $Directory
8170ee30     4     36     4     0  AppEvent.Evt
816223f8     4      0     0     0  $Directory
8170ec28     8     28     4     0  SecEvent.Evt
816220a8     4      0     0     0  $Directory
8170ea20     4     32     4     0  SysEvent.Evt
8170d188   232      0     0     0  NTUSER.DAT
81709f10     8      0     0     0  UsrClass.dat
81708918   232      0     0     0  NTUSER.DAT
81708748     8      0     0     0  UsrClass.dat
816c58f8    12      0     0     0  change.log
815c3880     4      0     0     0  $Directory
81706aa8     4      0     0     0  SchedLgU.Txt
815ba2d8     4      0     0     0  $Directory
815aa5f8     8      0     0     0  $Directory
8166d728    44      0     0     0  Netlogon.log
81701120     8     16     4     0  es.dll
816ff0a8     4      8     4     0  stdole2.tlb
8159a358     4      0     0     0  $Directory
8159da70     4      0     0     0  $Directory
8159c158     4      0     0     0  $Directory
815cb9b0     4      0     0     0  00000001
81779b20     4      0     0     0  $Directory
8159ac20     4      0     0     0  $Directory
815683f8     4      0     0     0  $Directory
81566978   580      0     0     0  NTUSER.DAT
81568460     4      0     0     0  $Directory
815675d8    68      0     0     0  UsrClass.dat
81567640     4      0     0     0  $Directory
...
81515878     4      0     0     0  $Directory
81516870     8      0     0     0  $Directory
8150df60     4      0     0     0  $Directory
...
816e5300     4      0     0     0  $Directory
8152afa0    16    212     0     0  msmsgs.exe
8153bbd8     4     32     0     0  stdole32.tlb
8172f950   488    392     0     0  OBJECTS.DATA
8173e9c0     4      0     0     0  $Directory
814f4538     4      0     0     0  $Directory
```

```
81650790   344     48      0      0  INDEX.BTR
814f55f8     4      0      0      0  $Directory
...
814caef8     4      0      0      0  $Directory
8171cd90  1392     36      0      0  system
815f07f0     4      0      0      0  $Directory
814a2298     4      0      0      0  $Directory
81541538     4      0      0      0  $Directory
81585288    28      0      0      0  $Directory
8173f708     4      0      0      0  $Directory
...
8158cf10     4      0      0      0  $Directory
```

!filelock

12/5/2018 • 2 minutes to read • Edit Online

The **!filelock** extension displays a file lock structure.

Syntax

```
!filelock FileLockAddress
!filelock ObjectAddress
```

Parameters

FileLockAddress
Specifies the hexadecimal address of the file lock structure.

ObjectAddress
Specifies the hexadecimal address of a file object that owns the file lock.

DLL

Kdexts.dll

Additional Information

For information about file objects, see the Windows Driver Kit (WDK) documentation and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

!fileobj

10/23/2019 • 2 minutes to read • Edit Online

The **!fileobj** extension displays detailed information about a FILE_OBJECT structure.

```
!fileobj FileObject
```

Parameters

FileObject
Specifies the address of a FILE_OBJECT structure.

DLL

Windows 2000	Unavailable
Windows XP and later	Kdexts.dll

Additional Information

For information about file objects, see the Microsoft Windows SDK documentation, the Windows Driver Kit (WDK) documentation, and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

If the FILE_OBJECT structure has an associated cache, **!fileobj** tries to parse and display cache information..

!filetime

12/5/2018 • 2 minutes to read • Edit Online

The **!filetime** extension converts a 64-bit FILETIME structure into a human-readable time.

```
!filetime Time
```

Parameters

Time
Specifies a 64-bit FILETIME structure.

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Kdexts.dll

Remarks

Here is an example of the output from this extension:

```
kd> !filetime 1c4730984712348
 7/26/2004 04:10:18.712 (Pacific Standard Time)
```

!finddata

8/26/2019 • 2 minutes to read • Edit Online

The **!finddata** extension displays the cached data at a given offset within a specified file object.

```
!finddata FileObject Offset
```

Parameters

FileObject
Specifies the address of the file object.

Offset
Specifies the offset.

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Kdexts.dll

Additional Information

For information about cache management, see the Microsoft Windows SDK documentation and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

For information about other cache management extensions, see the **!cchelp** extension.

!findfilelockowner

8/26/2019 • 2 minutes to read • Edit Online

The **!findfilelockowner** extension attempts to find the owner of a file object lock by examining all threads for a thread that is blocked in an IopSynchronousServiceTail assert and that has the file object as a parameter.

```
!findfilelockowner [FileObject]
```

Parameters

FileObject
Specifies the address of a file object. If *FileObject* is omitted, the extension searches for any thread in the current process that is stuck in **IopAcquireFileObjectLock** and retrieves the file object address from the stack trace.

DLL

Windows 2000	Unavailable
Windows XP and later	Kdexts.dll

Additional Information

For information about file objects, see the Microsoft Windows SDK documentation, the Windows Driver Kit (WDK) documentation, and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

This extension is most useful after a critical section timeout in which the thread that times out was waiting for the file inside **IopAcquireFileObjectLock**. After the offending thread is found, the extension attempts to recover the IRP that was used for the request and to display the driver that was processing the IRP.

The extension takes some time to complete because it walks the stack of all threads in the system until it finds the offending thread.You can stop ` at any point by pressing CTRL+BREAK (in WinDbg) or CTRL+C (in KD).

!findthreads

4/8/2019 • 2 minutes to read • Edit Online

The !findthreads extension displays summary information about one or more threads on the target system based on supplied search criteria. Thread information will be displayed when the associated stack(s) reference the supplied object. This command can be used only during kernel-mode debugging.

Syntax

```
!findthreads [-v][-t  <Thread Address>|-i <IRP Address>|-d <Device Address>|( -a <Pointer Address> -e <End Address> | -l <Range Length>)]
```

Parameters

-v
Displays verbose information on all criteria matches.

-t **** *Thread Address*
The search criteria will be all modules, wait objects, and IRPs for the thread, as well as device objects and modules generated from the attached IRPs. This option will generally provide the broadest search criteria.

-i **** *IRP Address*
The search criteria will be all modules and devices for the indicated IRP, as well as any reference to the IRP itself.

-d **** *Device Address*
The search criteria will be based from the device object. This will include the module associated with the device object (through the driver object), the device extension, any IRP attached to the device, and similar information for any attached to the device object.

-a **** *Pointer Address*
Add a base address to the criteria. If -e or -l is correctly specified, this value will be the base of a memory range. Otherwise it is interpreted as a pointer.

-e **** *End Address*
Specifies the end address of the memory range specified with -a.

-l **** *Range Length*
Specifies the length of the memory range specified with -a.

DLL

Windows 10 and later	Kdexts.dll

Additional Information

For information about threads in kernel mode, see Changing Contexts. For more information about analyzing processes and threads, see *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

Here is example output using the -v and -t options:

```
kd> !findthreads -v -t ffffd001ee29cdc0

Added criterion for THREAD 0xffffd001ee29cdc0
  Added criterion for THREAD STACK 0xffffd001ee2bac20
  ERROR: Object 0xfffffffffffffe0 is not an IRP
ERROR: unable to completely walk thread IRP list.
  Added criterion for MODULE kdnic(0xfffff80013120000)

Found 63 threads matching the search criteria

Found 6 criteria matches for THREAD 0xffffe0016a383740, PROCESS 0xffffe0016a220200
  Kernel stack location 0xffffd001f026a0c0 references THREAD 0xffffd001ee29cdc0
  Kernel stack location 0xffffd001f026a418 references THREAD 0xffffd001ee29cdc0
  Kernel stack location 0xffffd001f026a460 references THREAD 0xffffd001ee29cdc0
  Kernel stack location 0xffffd001f026a4d0 references THREAD 0xffffd001ee29cdc0
  Kernel stack location 0xffffd001f026a4f0 references THREAD 0xffffd001ee29cdc0
  Kernel stack location 0xffffd001f026a670 references THREAD 0xffffd001ee29cdc0


    ffffd001f026a0e0 nt!KiSwapContext+76
    ffffd001f026a190 nt!KiSwapThread+1c8
    ffffd001f026a220 nt!KiCommitThreadWait+148
    ffffd001f026a2e0 nt!KeWaitForMultipleObjects+21e
    ffffd001f026a800 nt!ObWaitForMultipleObjects+2b7
    ffffd001f026aa80 nt!NtWaitForMultipleObjects+f6
    000000c8d220fa98 nt!KiSystemServiceCopyEnd+13
    000000c8d220fa98 ntdll!ZwWaitForMultipleObjects+a
...
```

!for_each_process

12/5/2018 • 2 minutes to read • Edit Online

The **!for_each_process** extension executes the specified debugger command once for each process in the target.

```
!for_each_process ["CommandString"]
!for_each_process -?
```

Parameters

CommandString
Specifies the debugger commands to be executed for each process.

If *CommandString* includes multiple commands, separate them with semicolons (;) and enclose *CommandString* in quotation marks ("). If *CommandString* is enclosed in quotations marks, the individual commands within *CommandString* cannot contain quotation marks. Within *CommandString*, **@#Process** is replaced by the process address.

-?
Displays help for this extension in the Debugger Command window.

DLL

This extension works only in kernel mode, even though it resides in Ext.dll.

Windows 2000	Ext.dll
Windows XP and later	Ext.dll

Additional Information

For general information about processes, see Threads and Processes. For information about manipulating or obtaining information about processes, see Controlling Processes and Threads.

Remarks

If no arguments are supplied, the debugger displays a list of all processes, along with time and priority statistics. This is equivalent to entering **!process @#Process 0** as the *CommandString* value.

To terminate execution at any point, press CTRL+BREAK (in WinDbg) or CTRL+C (in KD).

!for_each_thread

12/5/2018 • 2 minutes to read • Edit Online

The **!for_each_thread** extension executes the specified debugger command once for each thread in the target.

```
!for_each_thread ["CommandString"]
!for_each_thread -?
```

Parameters

CommandString
Specifies the debugger commands to be executed for each thread. If *CommandString* includes multiple commands, separate them with semicolons (;) and enclose *CommandString* in quotation marks ("). If *CommandString* is enclosed in quotations marks, the individual commands within *CommandString* cannot contain quotation marks. Within *CommandString*, **@#Thread** is replaced by the thread address.

-?
Displays help for this extension in the Debugger Command window.

DLL

This extension works only in kernel mode, even though it resides in Ext.dll.

Windows 2000	Ext.dll
Windows XP and later	Ext.dll

Additional Information

For more general information about threads, see Threads and Processes. For more information about manipulating or obtaining information about threads, see Controlling Processes and Threads.

Remarks

If no arguments are supplied, the debugger displays a list of all threads, along with thread wait states. This is equivalent to entering **!thread @#Thread 2** as the *CommandString* value.

You can terminate execution at any point by pressing CTRL+BREAK (in WinDbg) or CTRL+C (in KD).

!fpsearch

12/5/2018 • 2 minutes to read • Edit Online

The **!fpsearch** extension searches the freed special pool for a specified address.

```
!fpsearch [Address] [Flag]
```

Parameters

Address
Specifies a virtual address.

Flag
If set, the debugger displays the raw content of each page on the free list as it searches the freed special pool.

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Kdexts.dll

Remarks

The display for an address includes the virtual address, the page frame number (PFN), the pool tag, size, whether the data at the address is pageable, the thread ID, and the call stack at the time of deallocation.

If *Address* is set to -1, the debugger displays the entire freed special pool.

If the debugger cannot find the specified address in the freed special pool, it does not display anything.Here is an example of the output from this extension:

```
kd> !fpsearch -1 1
Searching the free page list (8 entries) for all freed special pool

1EC4000  04000200 e56b6f54 000001f4 0000059c ....Tok.........
1EC4000  00000800 00000000 00000000 00000000 ................
1EC4000  bad0b0b0 82100000 00000000 00000000 ................
1EC4000  72657355 20203233 0000bac5 00000000 User32  ........
1EC4000  00028b94 00000000 0000bac9 00000000 ................
1EC4000  00000000 00000000 ffffffff 7fffffff ................
1EC4000  8153b1b8 00028aff 00000000 00000000 ..S.............
1EC4000  0000001b 00000000 00000012 00000514 ................

26A2000  000a0008 00adecb0 000e000c 00adecba ................
26A2000  000a0008 00adecc8 000e000c 00adecd2 ................
26A2000  000e000c 00adece0 000e000c 00adecee ................
26A2000  00120010 00adecfc 000e000c 00aded0e ................
26A2000  000e000c 00aded1c 000e000c 00aded2a ............*...
26A2000  000e000c 00aded38 000e000c 00aded46 ....8.......F...
26A2000  000a0008 00aded54 000e000c 00aded5e ....T.......^...
26A2000  00120010 00aded6c 000e000c 00aded7e ....l.......~...

2161000  000a0008 00adeccc 000e000c 00adecd6 ................
2161000  000a0008 00adece4 000e000c 00adecee ................
2161000  000e000c 00adecfc 000e000c 00aded0a ................
2161000  00120010 00aded18 000e000c 00aded2a ............*...
2161000  000e000c 00aded38 000e000c 00aded46 ....8.......F...
2161000  000e000c 00aded54 000e000c 00aded62 ....T.......b...
2161000  000a0008 00aded70 000e000c 00aded7a ....p.......z...
2161000  00120010 00aded88 000e000c 00aded9a ................

...

CEC8000  0311ffa4 03120000 0311c000 00000000 ................
CEC8000  00001e00 00000000 7ff88000 00000000 ................
CEC8000  00000328 00000704 00000000 00000000 (...............
CEC8000  7ffdf000 00000000 00000000 00000000 ................
CEC8000  e18ba8c0 00000000 00000000 00000000 ................
CEC8000  00000000 00000000 00000000 00000000 ................
CEC8000  00000000 00000000 00000000 00000000 ................
CEC8000  00000000 00000000 00000000 00000000 ................

CEAD000  00000000 00000000 00000000 00000000 ................
CEAD000  00000000 00000000 00000000 00000000 ................
CEAD000  00000000 00000000 00000000 00000000 ................
CEAD000  00000000 00000000 00000000 00000000 ................
CEAD000  00000000 00000000 00000000 00000000 ................
CEAD000  00000000 00000000 00000000 00000000 ................
CEAD000  00000000 00000000 00000000 00000000 ................
CEAD000  00000000 00000000 00000000 00000000 ................
```

You can stop execution at any point by pressing CTRL+BREAK (in WinDbg) or CTRL+C (in KD).

!frozen

12/5/2018 • 2 minutes to read • Edit Online

The **!frozen** extension displays the state of each processor.

```
!frozen
```

DLL

Windows 2000	Unavailable
Windows XP and later	Kdexts.dll

Remarks

Here is an example of the output from this extension:

```
0: kd> !frozen
Processor states:
       0 : Current
       1 : Frozen
```

!fwver

12/5/2018 • 2 minutes to read • Edit Online

The **!fwver** extension displays the Itanium firmware version.

```
!fwver
```

Important This command has been deprecated in the Windows Debugger Version 10.0.14257 and later, and is no longer available.

DLL

Windows 2000	Unavailable
Windows XP and later	Kdexts.dll

This extension command can only be used with an Itanium target computer.

Additional Information

For more information, consult an Intel architecture manual.

Remarks

Here is an example of the output from this extension:

```
kd> !fwver

Firmware Version

   Sal Revision:        0
   SAL_A_VERSION:       0
   SAL_B_VERSION:       0
   PAL_A_VERSION:       6623
   PAL_B_VERSION:       6625
   smbiosString:        W460GXBS2.86E.0117A.P08.200107261041
```

!fxdevice

6/25/2019 • 2 minutes to read • Edit Online

The !fxdevice extension displays summary information about all Power Management Framework (PoFx) registered devices. This command can be used only during kernel-mode debugging.

For more information about PoFX, see Overview of the Power Management Framework.

Syntax

```
!fxdevice[<FxDevice Address>]
```

Parameters

< FxDevice Address>
Provides the address of the FxDevice to display.

DLL

Windows 10 and later	Kdexts.dll

Remarks

The !fxdevice extension displays the following information when it is present on the target system.

- Non-idle PoFx devices
- Idle D0 PoFx devices
- Idle non-D0 PoFx devices

The following is example out from the !fxdevice extension with a supplied device address.

```
kd> !fxdevice ffffe0012ccbda60
!fxdevice 0xffffe0012ccbda60
    DevNode: 0xffffe0012bbb09f0
    UniqueId: "HDAUDIO\FUNC_01&VEN_10EC&DEV_0662&SUBSYS_103C304A&REV_1001\4&25ff998c&0&0001"
    InstancePath: "HDAUDIO\FUNC_01&VEN_10EC&DEV_0662&SUBSYS_103C304A&REV_1001\4&25ff998c&0&0001"
    Device Power State: PowerDeviceD0
    PEP Owner: Default PEP
    Acpi Plugin: 0
    Acpi Handle: 0
    Device Status Flags: IdleTimerOn DevicePowerNotRequired_ReceivedFromPEP
    Device Idle Timeout: 0x1869ffffe7960
    Device Power On: No Activity
    Device Power Off: No Activity
    Device Unregister: No Activity
    Component Count: 1
        Component 0: F0/F0 - IDLE   (RefCount = 0)
        Pep Component: 0xffffe0012cfe1800
            Active: 0   Latency: 0  Residency: 0    Wake: 0 Dx IRP: 0   WW IRP: 0
            Component Idle State Change: No Activity
            Component Activation: No Activity
            Component Active: No Activity
```

The following is the default output from the !fxdevice extension.

```
kd> !fxdevice
*******************************************************************************
Dumping non-idle PoFx devices
*******************************************************************************
!fxdevice 0xffffe0012bbd08d0
    DevNode: 0xffffe0012b3f87b0
    UniqueId: "\_SB.PCI0"
    InstancePath: "ACPI\PNP0A08\2&daba3ff&1"
    Device Power State: PowerDeviceD0
    Component Count: 1
        Component 0: F0/F1 - ACTIVE (RefCount = 28)

!fxdevice 0xffffe0012c587940
    DevNode: 0xffffe0012b3f9d30
    UniqueId: "\_SB.PCI0.PEGL"
    InstancePath: "PCI\VEN_8086&DEV_D138&SUBSYS_304A103C&REV_11\3&33fd14ca&0&18"
    Device Power State: PowerDeviceD0
    Component Count: 1
        Component 0: F0/F1 - ACTIVE (RefCount = 1)

...

*******************************************************************************
Dumping idle D0 PoFx devices
*******************************************************************************
!fxdevice 0xffffe0012c5838c0
    DevNode: 0xffffe0012bbdfd30
    UniqueId: "\_SB.PCI0.PCX1"
    InstancePath: "PCI\VEN_8086&DEV_3B42&SUBSYS_304A103C&REV_05\3&33fd14ca&0&E0"
    Device Power State: PowerDeviceD0
    Component Count: 1
        Component 0: F1/F1 - IDLE   (RefCount = 0)

!fxdevice 0xffffe0012c581ac0
    DevNode: 0xffffe0012bbdfa50
    UniqueId: "\_SB.PCI0.PCX5"
    InstancePath: "PCI\VEN_8086&DEV_3B4A&SUBSYS_304A103C&REV_05\3&33fd14ca&0&E4"
    Device Power State: PowerDeviceD0
    Component Count: 1
        Component 0: F1/F1 - IDLE   (RefCount = 0)

...
```

!gbl

8/26/2019 • 2 minutes to read • Edit Online

The **!gbl** extension displays header information from the ACPI BIOS Root System Description (RSDT) table of the target computer.

```
!gbl [-v]
```

Parameters

-v
Verbose. Displays detailed information about the table.

DLL

Windows 2000	Unavailable
Windows XP and later	Kdexts.dll

Additional Information

For information about the ACPI and ACPI tables, see Other ACPI Debugging Extensions and the ACPI Specification Web site. Also see the Microsoft Windows SDK documentation, the Windows Driver Kit (WDK) documentation, and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

!gentable

12/5/2018 • 2 minutes to read • Edit Online

The **!gentable** extension displays an RTL_GENERIC_TABLE.

Syntax

```
!gentable Address[Flag]
```

Parameters

Address
Specifies the address of the RTL_GENERIC_TABLE.

Flag
Specifies the table source. If *Flag* is 1, the AVL table is used. If *Flag* is 0 or omitted, the non-AVL table is used.

DLL

Kdexts.dll

!hidppd

12/5/2018 • 2 minutes to read • Edit Online

The **!hidppd** extension displays the contents of the HIDP_PREPARSED_DATA structure.

```
!hidppd Address
```

Parameters

Address
Specifies the hexadecimal address of the HIDP_PREPARSED_DATA structure.

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Kdexts.dll

Additional Information

For information about human input devices (HID), see the Windows Driver Kit (WDK) documentation.

!ib, !id, !iw

12/5/2018 • 2 minutes to read • Edit Online

The **!ib**, **!id**, and **!iw** extension commands are obsolete. Use the **ib, id, iw (Input from Port)** commands instead.

!icpleak

6/25/2019 • 2 minutes to read • Edit Online

The **!icpleak** extension examines all I/O completion objects in the system for the object with the largest number of queued entries.

```
!icpleak [HandleFlag]
```

Parameters

HandleFlag
If this flag is set, the display also includes all processes that have a handle to the object with the largest number of queued entries.

DLL

Windows 2000	Unavailable
Windows XP and later	Kdexts.dll

Additional Information

For information about I/O completion ports, see *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

This extension is useful when there is a leak in the I/O completion pool. I/O completion pool leaks can occur when a process is allocating I/O completion packets by calling **PostQueuedCompletionStatus**, but is not calling **GetQueuedCompletionStatus** to free them, or when a process is queuing completion entries to a port, but there is no thread retrieving the entries. To detect a leak run the **!poolused** extension and check the value of ICP pool tag. If pool use with the ICP tag is significant, a leak might have occurred.

This extension works only if the system maintains type lists. If the *HandleFlag* is set and the system has many processes, this extension will take a long time to run.

You can stop at any point by pressing CTRL+BREAK (in WinDbg) or CTRL+C (in KD).

!idt

8/26/2019 • 2 minutes to read • Edit Online

The **!idt** extension displays the interrupt service routines (ISRs) for a specified interrupt dispatch table (IDT).

```
!idt IDT
!idt [-a]
!idt -?
```

Parameters

IDT
Specifies the IDT to display.

-a
When *IDT* is not specified, the debugger displays the IDTs of all processors on the target computer in an abbreviated format. If **-a** is specified, the ISRs for each IDT are also displayed.

-?
Displays help for this extension in the Debugger Command window.

DLL

Windows 2000	Unavailable
Windows XP and later	Kdexts.dll

This extension command can only be used with an x64-based or x86-based target computer.

Additional Information

For information about ISRs and IDTs, see the Windows Driver Kit (WDK) documentation and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

For more information about how to display the interrupt vector tables on an Itanium target computer, see **!ivt**.

Remarks

Here is an example of the output from this extension:

```
0: kd> !idt

Dumping IDT:

37:806ba78c hal!PicSpuriousService37
3d:806bbc90 hal!HalpApcInterrupt
41:806bbb04 hal!HalpDispatchInterrupt
50:806ba864 hal!HalpApicRebootService
63:8641376c VIDEOPRT!pVideoPortInterrupt (KINTERRUPT 86413730)
73:862aa044 portcls!CInterruptSyncServiceRoutine (KINTERRUPT 862aa008)
82:86594314 atapi!IdePortInterrupt (KINTERRUPT 865942d8)
83:86591bec SCSIPORT!ScsiPortInterrupt (KINTERRUPT 86591bb0)
92:862b53dc serial!SerialCIsrSw (KINTERRUPT 862b53a0)
93:86435844 i8042prt!I8042KeyboardInterruptService (KINTERRUPT 86435808)
a3:863b366c i8042prt!I8042MouseInterruptService (KINTERRUPT 863b3630)
a4:8636bbec USBPORT!USBPORT_InterruptService (KINTERRUPT 8636bbb0)
b1:86585bec ACPI!ACPIInterruptServiceRoutine (KINTERRUPT 86585bb0)
b2:863c0524 serial!SerialCIsrSw (KINTERRUPT 863c04e8)
b4:86391a54 NDIS!ndisMIsr (KINTERRUPT 86391a18)
         USBPORT!USBPORT_InterruptService (KINTERRUPT 863ae890)
c1:806ba9d0 hal!HalpBroadcastCallService
d1:806b9dd4 hal!HalpClockInterrupt
e1:806baf30 hal!HalpIpiHandler
e3:806baca8 hal!HalpLocalApicErrorService
fd:806bb460 hal!HalpProfileInterrupt
```

!ih

12/5/2018 • 2 minutes to read • Edit Online

The **!ih** extension displays the interrupt history record for the specified processor.

```
!ih Processor
```

Important This command has been deprecated in the Windows Debugger Version 10.0.14257 and later, and is no longer available.

Parameters

Processor
Specifies a processor. If *Processor* is omitted, the current processor is used.

DLL

Windows 2000	Unavailable
Windows XP and later	Kdexts.dll

This extension command can only be used with an Itanium-based target computer.

Remarks

This extension displays the interrupt history record without referencing the program counter symbols. To display the interrupt history record using the program counter symbols, use the **!ihs** extension. To enable the interrupt history record, add **/configflag=32** to the boot entry options.

Here is an example of the output from this extension:

```
kd> !ih
Total # of interruptions = 2093185
Vector              IIP                   IPSR          ExtraField
 VHPT FAULT  e0000000830d3190      1010092a6018  IFA=      6fc00a0200c
        VHPT FAULT  e0000000830d33d0      1010092a6018  IFA= 1ffffe00001de2d0
 VHPT FAULT  e0000000830d33d0      1010092a6018  IFA= 1ffffe01befff338
        VHPT FAULT  e0000000830d3190      1010092a6018  IFA=      6fc00a0200c
        VHPT FAULT  e0000000830d33d0      1010092a6018  IFA= 1ffffe00001d9188
        VHPT FAULT  e0000000830d3880      1010092a6018  IFA= 1ffffe01befff250
        VHPT FAULT  e0000000830d3fb0      1010092a6018  IFA= e0000165f82dc1c0
 VHPT FAULT  e000000083063390      1010092a6018  IFA= e0000000fffe0018
     THREAD SWITCH  e000000085896040  e00000008588c040  OSP= e0000165f82dbd40
        VHPT FAULT  e00000008329b130      1210092a6018  IFA= e0000165f7edaf30
        VHPT FAULT  e0000165f7eda640      1210092a6018  IFA= e0000165fff968a9
    PROCESS SWITCH  e0000000818bbe10  e000000085896040  OSP= e0000165f8281de0
        VHPT FAULT  e00000008307cfc0      1010092a2018  IFA= e00000008189fe50
EXTERNAL INTERRUPT  e0000000830623b0      1010092a6018  IVR=               d0
        VHPT FAULT  e00000008314e310      1010092a2018  IFA= e0000165f88203f0
        VHPT FAULT  e000000083580760      1010092a2018  IFA= e0000000f00ff3fd
    PROCESS SWITCH  e00000008558c990  e0000000818bbe10  OSP= e00000008189fe20
        VHPT FAULT  e00000008307cfc0      1010092a2018  IFA= e0000165f02433f0
        VHPT FAULT          77cfbda0      1013082a6018  IFA=         77cfbda0
 VHPT FAULT          77cfbdb0      1213082a6018  IFA=      6fbfee0ff98
   DATA ACCESS BIT          77b8e150      1213082a6018  IFA=         77c16ab8
        VHPT FAULT          77ed5d60      1013082a6018  IFA=      6fbfffa6048
   DATA ACCESS BIT          77ed5d60      1213082a6018  IFA=         77c229c0
 DATA ACCESS BIT          77b8e1b0      1013082a6018  IFA=         77c1c320
      USER SYSCALL          77cafa40        10082a6018  Num=               42
 VHPT FAULT  e00000008344dc20      1010092a6018  IFA= e000010600703784
...
```

!ihs

12/5/2018 • 2 minutes to read • Edit Online

The **!ihs** extension displays the interrupt history record for the specified processor, using program counter symbols.

```
!ihs Processor
```

Important This command has been deprecated in the Windows Debugger Version 10.0.14257 and later, and is no longer available.

Parameters

Processor
Specifies a processor. If *Processor* is omitted, the current processor is used.

DLL

Windows 2000	Unavailable
Windows XP and later	Kdexts.dll

This extension command can only be used with an Itanium target computer.

Remarks

To display the interrupt history record without using program counter symbols, use the **!ih** extension. To enable the interrupt history record, add **/configflag=32** to the boot entry options.

Here is an example of the output from this extension:

```
kd> !ihs
Total # of interruptions = 2093185
Vector              IIP                  IPSR          ExtraField            IIP Symbol
        VHPT FAULT  e0000000830d3190      1010092a6018  IFA=      6fc00a0200c
nt!MiAgeAndEstimateAvailableInWorkingSet+0x70
        VHPT FAULT  e0000000830d33d0      1010092a6018  IFA= 1ffffe00001de2d0
nt!MiAgeAndEstimateAvailableInWorkingSet+0x2b0
        VHPT FAULT  e0000000830d33d0      1010092a6018  IFA= 1ffffe01befff338
nt!MiAgeAndEstimateAvailableInWorkingSet+0x2b0
        VHPT FAULT  e0000000830d3190      1010092a6018  IFA=      6fc00a0200c
nt!MiAgeAndEstimateAvailableInWorkingSet+0x70
        VHPT FAULT  e0000000830d33d0      1010092a6018  IFA= 1ffffe00001d9188
nt!MiAgeAndEstimateAvailableInWorkingSet+0x2b0
        VHPT FAULT  e0000000830d3880      1010092a6018  IFA= 1ffffe01befff250
nt!MiAgeAndEstimateAvailableInWorkingSet+0x760
        VHPT FAULT  e0000000830d3fb0      1010092a6018  IFA= e0000165f82dc1c0
nt!MiCheckAndSetSystemTrimCriteria+0x190
        VHPT FAULT  e000000083063390      1010092a6018  IFA= e0000000fffe0018 nt!KeQuerySystemTime+0x30
     THREAD SWITCH  e000000085896040  e00000008588c040  OSP= e0000165f82dbd40
        VHPT FAULT  e000000083290130      1210092a6018  IFA= e0000165f7edaf30 nt!IopProcessWorkItem+0x30
 VHPT FAULT  e0000165f7eda640      1210092a6018  IFA= e0000165fff968a9 netbios!RunTimerForLana+0x60
    PROCESS SWITCH  e0000000818bbe10  e000000085896040  OSP= e0000165f8281de0
        VHPT FAULT  e00000008307cfc0      1010092a2018  IFA= e00000008189fe50 nt!SwapFromIdle+0x1e0
EXTERNAL INTERRUPT  e0000000830623b0      1010092a6018  IVR=               d0 nt!Kil_TopOfIdleLoop
        VHPT FAULT  e00000008314e310      1010092a2018  IFA= e0000165f88203f0 nt!KdReceivePacket+0x10
        VHPT FAULT  e000000083580760      1010092a2018  IFA= e0000000f00ff3fd hal!READ_PORT_UCHAR+0x80
    PROCESS SWITCH  e00000008558c990  e0000000818bbe10  OSP= e00000008189fe20
        VHPT FAULT  e00000008307cfc0      1010092a2018  IFA= e0000165f02433f0 nt!SwapFromIdle+0x1e0
        VHPT FAULT          77cfbda0      1013082a6018  IFA=         77cfbda0 0x0000000077cfbda0
        VHPT FAULT          77cfbdb0      1213082a6018  IFA=      6fbfee0ff98 0x0000000077cfbdb0
 DATA ACCESS BIT          77b8e150      1213082a6018  IFA=         77c16ab8 0x0000000077b8e150
 VHPT FAULT          77ed5d60      1013082a6018  IFA=      6fbfffa6048 0x0000000077ed5d60
   DATA ACCESS BIT          77ed5d60      1213082a6018  IFA=         77c229c0 0x0000000077ed5d60
 DATA ACCESS BIT          77b8e1b0      1013082a6018  IFA=         77c1c320 0x0000000077b8e1b0
      USER SYSCALL          77cafa40        10082a6018  Num=               42 0x0000000077cafa40
 VHPT FAULT  e00000008344dc20      1010092a6018  IFA= e000010600703784 nt!ExpLookupHandleTableEntry+0x20
...
```

!ioresdes

12/5/2018 • 2 minutes to read • Edit Online

The **!ioresdes** extension displays the IO_RESOURCE_DESCRIPTOR structure at the specified address.

```
!ioresdes Address
```

Parameters

Address
Specifies the hexadecimal address of the IO_RESOURCE_DESCRIPTOR structure.

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Kdexts.dll

Additional Information

See Plug and Play Debugging for applications of this extension command. For information about the IO_RESOURCE_DESCRIPTOR structure, see the Windows Driver Kit (WDK) documentation.

!ioctldecode

10/23/2019 • 2 minutes to read • Edit Online

The **!ioctldecode** extension displays the *Device Type*, *Required Access*, *Function Code* and *Transfer Type* as specified by the given IOCTL code. For more information about IOCTL control codes, see Defining I/O Control Codes.

```
!ioctldecode IoctlCode
```

Parameters

IoctlCode
Specifies the hexadecimal IOCTL Code. The **!irp** command displays the IOCTL code in its output.

DLL

Kdexts.dll

Additional Information

To see information on the IOCTL, we first locate an IRP of interest. You can use the **!irpfind** command to locate an irp of interest.

Use the **!irp** command to display information about the irp.

```
0: kd> !irp ffffd581a6c6cd30
Irp is active with 6 stacks 6 is current (= 0xffffd581a6c6cf68)
No Mdl: No System Buffer: Thread 00000000:  Irp stack trace.
     cmd  flg cl Device   File     Completion-Context
[N/A(0), N/A(0)]
            0  0 00000000 00000000 00000000-00000000

                                        Args: 00000000 00000000 00000000 00000000
[N/A(0), N/A(0)]
            0  0 00000000 00000000 00000000-00000000

                                        Args: 00000000 00000000 00000000 00000000
[N/A(0), N/A(0)]
            0  0 00000000 00000000 00000000-00000000

                                        Args: 00000000 00000000 00000000 00000000
[N/A(0), N/A(0)]
            0  0 00000000 00000000 00000000-00000000

                                        Args: 00000000 00000000 00000000 00000000
[N/A(0), N/A(0)]
            0  0 00000000 00000000 00000000-00000000

                                        Args: 00000000 00000000 00000000 00000000
>[IRP_MJ_INTERNAL_DEVICE_CONTROL(f), N/A(0)]
            0 e1 ffffd581a5fbd050 00000000 fffff806d2412cf0-ffffd581a5cce050 Success Error Cancel pending
                       \Driver\usbehci        (IopUnloadSafeCompletion)
                                        Args: ffffd581a6c61a50 00000000 0x220003 00000000
```

The third argument displayed, in this case *0x220003*, is the IOCTL code. Use the IOCTL code to display

information about the IOCTL, in this case **IOCTL_INTERNAL_USB_SUBMIT_URB**.

```
0: kd> !ioctldecode 0x220003

IOCTL_INTERNAL_USB_SUBMIT_URB

Device Type    : 0x22 (FILE_DEVICE_WINLOAD) (FILE_DEVICE_USER_MODE_BUS) (FILE_DEVICE_USB)
(FILE_DEVICE_UNKNOWN)
Method         : 0x3 METHOD_NEITHER
Access         : FILE_ANY_ACCESS
Function       : 0x0
```

If you provide an IOCTL code that is not available, you will see this type of output.

```
0: kd> !ioctldecode 0x1280ce

Unknown IOCTL  : 0x1280ce

Device Type    : 0x12 (FILE_DEVICE_NETWORK)
Method         : 0x2 METHOD_OUT_DIRECT
Access         : FILE_WRITE_ACCESS
Function       : 0x33
```

Although the IOCTL is not identified, information about the IOCTL fields are displayed.

Note that only a subset of publicly defined IOCTLs are able to be identified by the **!ioctldecode** command.

For more information about IOCTLs see Introduction to I/O Control Codes.

For more general information about IRPs and IOCTLs, refer to *Windows Internals* by Mark E. Russinovich, David A. Solomon and Alex Ionescu.

!ioreslist

12/5/2018 • 2 minutes to read • Edit Online

The **!ioreslist** extension displays an IO_RESOURCE_REQUIREMENTS_LIST structure.

```
!ioreslist Address
```

Parameters

Address
Specifies the hexadecimal address of the IO_RESOURCE_REQUIREMENTS_LIST structure.

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Kdexts.dll

Additional Information

See Plug and Play Debugging for applications of this extension command. For information about the IO_RESOURCE_REQUIREMENTS_LIST structure, see the Windows Driver Kit (WDK) documentation.

Remarks

Here is an example of the output from this extension:

```
kd> !ioreslist 0xe122b768

IoResList at 0xe122b768 : Interface 0x5  Bus 0  Slot 0xe
  Alternative 0 (Version 1.1)
    Preferred Descriptor 0 - Port (0x1) Device Exclusive (0x1)
      Flags (0x01) - PORT_IO
      0x000100 byte range with alignment 0x000100
      1000 - 0x10ff
    Alternative Descriptor 1 - Port (0x1) Device Exclusive (0x1)
      Flags (0x01) - PORT_IO
      0x000100 byte range with alignment 0x000100
      0 - 0xffffffff
    Descriptor 2 - DevicePrivate (0x81) Device Exclusive (0x1)
      Flags (0000) -
      Data:              : 0x1 0x0 0x0
    Preferred Descriptor 3 - Memory (0x3) Device Exclusive (0x1)
      Flags (0000) - READ_WRITE
      0x001000 byte range with alignment 0x001000
      40080000 - 0x40080fff
    Alternative Descriptor 4 - Memory (0x3) Device Exclusive (0x1)
      Flags (0000) - READ_WRITE
      0x001000 byte range with alignment 0x001000
      0 - 0xffffffff
    Descriptor 5 - DevicePrivate (0x81) Device Exclusive (0x1)
      Flags (0000) -
      Data:              : 0x1 0x1 0x0
    Descriptor 6 - Interrupt (0x2) Shared (0x3)
      Flags (0000) - LEVEL_SENSITIVE
      0xb - 0xb
```

The IO_RESOURCE_REQUIREMENTS_LIST contains information about:

- Resource types

  There are four types of resources: I/O, Memory, IRQ, DMA.

- Descriptors

  Each preferred setting has a "Preferred" descriptor and a number of "Alternative" descriptors.

This resource list contains the following requests:

- I/O Ranges

  Prefers a range of 0x1000 to 0x10FF inclusive, but can use any 0x100 range between 0 and 0xFFFFFFFF, provided it is 0x100-aligned. (For example, 0x1100 to 0x11FF is acceptable.)

- Memory

  Prefers a range of 0x40080000 to 0x40080FFF, but can use any range that is of size 0x1000, is 0x1000-aligned, and is located between 0 and 0xFFFFFFFF.

- IRQ

  Must use IRQ 0xB.

Interrupts and DMA channels are represented as ranges with the same beginning and end.

!iovirp

12/5/2018 • 2 minutes to read • Edit Online

The **!iovirp** extension displays detailed information for a specified I/O Verifier IRP.

```
!iovirp [IRP]
```

Parameters

IRP
Specifies the address of an IRP tracked by the Driver Verifier. If *IRP* is 0 or is omitted, the summary information for each outstanding IRP is displayed.

DLL

Windows 2000	Unavailable
Windows XP and later	Kdexts.dll

Remarks

Here is an example of the output from this extension:

```
kd> !iovirp 947cef68
IovPacket       84509af0
TrackedIrp      947cef68
HeaderLock      84509d61
LockIrql        0
ReferenceCount  1
PointerCount    1
HeaderFlags     00000000
ChainHead       84509af0
Flags           00200009
DepartureIrql   0
ArrivalIrql     0
StackCount      1
QuotaCharge     00000000
QuotaProcess    0
RealIrpCompletionRoutine        0
RealIrpControl                  0
RealIrpContext                  0
TopStackLocation        2
PriorityBoost           0
LastLocation            0
RefTrackingCount        0
SystemDestVA            0
VerifierSettings        84509d08
pIovSessionData         84509380
Allocation Stack:
  nt!IovAllocateIrp+1a  (817df356)
 nt!IopXxxControlFile+40c  (8162de20)
  nt!NtDeviceIoControlFile+2a  (81633090)
 nt!KiFastCallEntry+164  (81513c64)
  nt!EtwpFlushBuffer+10f  (817606d7)
  nt!EtwpFlushBuffersWithMarker+bd  (817608cb)
  nt!EtwpFlushActiveBuffers+2b4  (81760bc2)
  nt!EtwpLogger+213  (8176036f)
```

You can stop execution at any point by pressing CTRL+BREAK (in WinDbg) or CTRL+C (in KD).

!ipi

12/5/2018 • 2 minutes to read • Edit Online

The **!ipi** extension displays the interprocessor interrupt (IPI) state for a specified processor.

```
!ipi [Processor]
```

Parameters

Processor
Specifies a processor. If *Processor* is omitted, the IPI state for every processor is displayed.

DLL

Windows 2000	Unavailable
Windows XP and later	Kdexts.dll

This extension command can only be used with an x86-based target computer.

Additional Information

For information about IPIs, see *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

Here is an example of the output from this extension:

```
0: kd> !ipi
IPI State for Processor 0
  Worker Routine:  nt!KiFlushTargetMultipleTb [Stale]
  Parameter[0]:    0
  Parameter[1]:    3
  Parameter[2]:    F7C98770
  Ipi Trap Frame:  F7CCCCDC [.trap F7CCCCDC]
  Signal Done:     0
  IPI Frozen:      24 [FreezeActive] [Owner]
  Request Summary: 0
  Target Set:      0
  Packet Barrier:  0

IPI State for Processor 1
  Worker Routine:  nt!KiFlushTargetMultipleTb [Stale]
  Parameter[0]:    1
  Parameter[1]:    3
  Parameter[2]:    F7CDCD28
  Ipi Trap Frame:  F7C8CCC4 [.trap F7C8CCC4]
  Signal Done:     0
  IPI Frozen:      2 [Frozen]
  Request Summary: 0
  Target Set:      0
  Packet Barrier:  0
```

!irp

10/23/2019 • 4 minutes to read • Edit Online

The **!irp** extension displays information about an I/O request packet (IRP).

```
!irp Address [Detail]
```

Parameters

Address
Specifies the hexadecimal address of the IRP.

Detail
If this parameter is included with any value, such as 1, the output includes the status of the IRP, the address of its memory descriptor list (MDL), its owning thread, and stack information for all of its I/O stacks, and information about each stack location for the IRP, including hexadecimal versions of the major function code and the minor function code. If this parameter omitted, the output includes only a summary of the information.

DLL

Windows XP and later	Kdexts.dll

Additional Information

See Plug and Play Debugging and Debugging Interrupt Storms for applications of this extension command. For information about IRPs, see the Windows Driver Kit (WDK) documentation and *Microsoft Windows Internals* by Mark Russinovich and David Solomon. For further information on the major and minor function codes, see the Windows Driver Kit (WDK) documentation.

This topic describes the IRP structure, **IRP**.

For detailed information on decoding the IRP structure including the returned Args, see the following resources.

- Windows Internals by Mark E. Russinovich, David A. Solomon and Alex Ionescu
- Developing Drivers with the Windows Driver Foundation Guy Smith and Penny Orwick

Remarks

The output also indicates under what conditions the completion routine for each stack location will be called once the IRP has completed and the stack location is processed. There are three possibilities:

Success
Indicates that the completion routine will be called when the IRP is completed with a success code.

Error
Indicates that the completion routine will be called when the IRP is completed with an error code.

Cancel
Indicates that the completion routine will be called if an attempt was made to cancel the IRP.

Any combination of these three may appear, and if any of the conditions shown are satisfied, the completion routine will be called. The appropriate values are listed at the end of the first row of information about each stack

location, immediately after the **Completion-Context** entry.

Here is an example of the output from this extension for Windows 10:

```
0: kd> !irp ac598dc8
Irp is active with 2 stacks 1 is current (= 0xac598e38)
 No Mdl: No System Buffer: Thread 8d1c7bc0:  Irp stack trace.
     cmd  flg cl Device   File     Completion-Context
>[IRP_MJ_FILE_SYSTEM_CONTROL(d), N/A(0)]
            1 e1 8a6434d8 ac598d40 853220cb-a89682d8 Success Error Cancel pending
           \FileSystem\Npfs fltmgr!FltpPassThroughCompletion
            Args: 00000000 00000000 00110008 00000000
 [IRP_MJ_FILE_SYSTEM_CONTROL(d), N/A(0)]
            1  0 8a799710 ac598d40 00000000-00000000
           \FileSystem\FltMgr
            Args: 00000000 00000000 0x00110008 00000000
```

Starting with Windows 10 the IRP major and minor code text is displayed, for example, "IRP_MJ_FILE_SYSTEM_CONTROL" The code value is also shown in hex, in this example "(d)".

The third argument displayed in the output, is the IOCTL code. Use the **!ioctldecode** command to display information about the IOCTL.

Here is an example of the output from this extension from Windows Vista.

```
0: kd> !irp 0x831f4a00
Irp is active with 8 stacks 5 is current (= 0x831f4b00)
 Mdl = 82b020d8 Thread 8c622118:  Irp stack trace.
     cmd  flg cl Device   File     Completion-Context
 [  0, 0]   0  0 00000000 00000000 00000000-00000000

                        Args: 00000000 00000000 00000000 00000000
 [  0, 0]   0  0 00000000 00000000 00000000-00000000

                        Args: 00000000 00000000 00000000 00000000
 [  0, 0]   0  0 00000000 00000000 00000000-00000000

                        Args: 00000000 00000000 00000000 00000000
 [  0, 0]   0  0 00000000 00000000 00000000-00000000

                        Args: 00000000 00000000 00000000 00000000
>[  3,34]  40 e1 828517a8 00000000 842511e0-00000000 Success Error Cancel pending
               \Driver\disk     partmgr!PmReadWriteCompletion
 Args: 00007000 00000000 fe084e00 00000004
 [  3, 0]  40 e0 82851450 00000000 842414d4-82956350 Success Error Cancel
 \Driver\PartMgr  volmgr!VmpReadWriteCompletionRoutine
                        Args: 129131bb 000000de fe084e00 00000004
 [  3, 0]   0 e0 82956298 00000000 847eeed0-829e2ba8 Success Error Cancel
 \Driver\volmgr   Ntfs!NtfsMasterIrpSyncCompletionRoutine
                        Args: 00007000 00000000 1bdae400 00000000
 [  3, 0]   0  0 82ac2020 8e879410 00000000-00000000
               \FileSystem\Ntfs
                        Args: 00007000 00000000 00018400 00000000
```

Note that the completion routine next to the driver name is set on that stack location, and it was set by the driver in the line below. In the preceding example, **Ntfs!NtfsMasterIrpSyncCompletionRoutine** was set by **\FileSystem\Ntfs**. The **Completion-Context** entry above **Ntfs!NtfsMasterIrpSyncCompletionRoutine**, **847eeed0-829e2ba8**, indicates the address of the completion routine, as well as the context that will be passed to **Ntfs!NtfsMasterIrpSyncCompletionRoutine**. From this we can see that the address of **Ntfs!NtfsMasterIrpSyncCompletionRoutine** is **847eeed0**, and the context that will be passed to this routine when it is called is **829e2ba8**.

IRP major function codes

The following information is included to help you interpret the output from this extension command.

The IRP major function codes are as follows:

MAJOR FUNCTION CODE	HEXADECIMAL CODE
IRP_MJ_CREATE	0x00
IRP_MJ_CREATE_NAMED_PIPE	0x01
IRP_MJ_CLOSE	0x02
IRP_MJ_READ	0x03
IRP_MJ_WRITE	0x04
IRP_MJ_QUERY_INFORMATION	0x05
IRP_MJ_SET_INFORMATION	0x06
IRP_MJ_QUERY_EA	0x07
IRP_MJ_SET_EA	0x08
IRP_MJ_FLUSH_BUFFERS	0x09
IRP_MJ_QUERY_VOLUME_INFORMATION	0x0A
IRP_MJ_SET_VOLUME_INFORMATION	0x0B
IRP_MJ_DIRECTORY_CONTROL	0x0C
IRP_MJ_FILE_SYSTEM_CONTROL	0x0D
IRP_MJ_DEVICE_CONTROL	0x0E
IRP_MJ_INTERNAL_DEVICE_CONTROL IRP_MJ_SCSI	0x0F
IRP_MJ_SHUTDOWN	0x10

MAJOR FUNCTION CODE	HEXADECIMAL CODE
IRP_MJ_LOCK_CONTROL	0x11
IRP_MJ_CLEANUP	0x12
IRP_MJ_CREATE_MAILSLOT	0x13
IRP_MJ_QUERY_SECURITY	0x14
IRP_MJ_SET_SECURITY	0x15
IRP_MJ_POWER	0x16
IRP_MJ_SYSTEM_CONTROL	0x17
IRP_MJ_DEVICE_CHANGE	0x18
IRP_MJ_QUERY_QUOTA	0x19
IRP_MJ_SET_QUOTA	0x1A
IRP_MJ_PNP IRP_MJ_MAXIMUM_FUNCTION	0x1B

The Plug and Play minor function codes are as follows:

MINOR FUNCTION CODE	HEXADECIMAL CODE
IRP_MN_START_DEVICE	0x00
IRP_MN_QUERY_REMOVE_DEVICE	0x01
IRP_MN_REMOVE_DEVICE	0x02
IRP_MN_CANCEL_REMOVE_DEVICE	0x03
IRP_MN_STOP_DEVICE	0x04
IRP_MN_QUERY_STOP_DEVICE	0x05

MINOR FUNCTION CODE	HEXADECIMAL CODE
IRP_MN_CANCEL_STOP_DEVICE	0x06
IRP_MN_QUERY_DEVICE_RELATIONS	0x07
IRP_MN_QUERY_INTERFACE	0x08
IRP_MN_QUERY_CAPABILITIES	0x09
IRP_MN_QUERY_RESOURCES	0x0A
IRP_MN_QUERY_RESOURCE_REQUIREMENTS	0x0B
IRP_MN_QUERY_DEVICE_TEXT	0x0C
IRP_MN_FILTER_RESOURCE_REQUIREMENTS	0x0D
IRP_MN_READ_CONFIG	0x0F
IRP_MN_WRITE_CONFIG	0x10
IRP_MN_EJECT	0x11
IRP_MN_SET_LOCK	0x12
IRP_MN_QUERY_ID	0x13
IRP_MN_QUERY_PNP_DEVICE_STATE	0x14
IRP_MN_QUERY_BUS_INFORMATION	0x15
IRP_MN_DEVICE_USAGE_NOTIFICATION	0x16
IRP_MN_SURPRISE_REMOVAL	0x17
IRP_MN_QUERY_LEGACY_BUS_INFORMATION	0x18

The WMI minor function codes are as follows:

MINOR FUNCTION CODE	HEXADECIMAL CODE
IRP_MN_QUERY_ALL_DATA	0x00
IRP_MN_QUERY_SINGLE_INSTANCE	0x01
IRP_MN_CHANGE_SINGLE_INSTANCE	0x02
IRP_MN_CHANGE_SINGLE_ITEM	0x03
IRP_MN_ENABLE_EVENTS	0x04
IRP_MN_DISABLE_EVENTS	0x05
IRP_MN_ENABLE_COLLECTION	0x06
IRP_MN_DISABLE_COLLECTION	0x07
IRP_MN_REGINFO	0x08
IRP_MN_EXECUTE_METHOD	0x09

The power management minor function codes are as follows:

MINOR FUNCTION CODE	HEXADECIMAL CODE
IRP_MN_WAIT_WAKE	0x00
IRP_MN_POWER_SEQUENCE	0x01
IRP_MN_SET_POWER	0x02
IRP_MN_QUERY_POWER	0x03

The SCSI minor function codes are as follows:

MINOR FUNCTION CODE	HEXADECIMAL CODE
IRP_MN_SCSI_CLASS	0x01

See also

IRP

!irpfind

!ioctldecode

!irpfind

12/5/2018 • 2 minutes to read • Edit Online

The **!irpfind** extension displays information about all I/O request packets (IRP) currently allocated in the target system, or about those IRPs matching the specified search criteria.

Syntax

```
!irpfind [-v][PoolType[RestartAddress[CriteriaData]]]
```

Parameters

-v
Displays verbose information.

PoolType
Specifies the type of pool to be searched. The following values are permitted:

0
Specifies nonpaged memory pool. This is the default.

1
Specifies paged memory pool.

2
Specifies the special pool.

4
Specifies the session pool.

RestartAddress
Specifies the hexadecimal address at which to begin the search. This is useful if the previous search was terminated prematurely. The default is zero.

Criteria
Specifies the criteria for the search. Only those IRPs that satisfy the given match will be displayed.

CRITERIA	MATCH
arg	Finds all IRPs with a stack location where one of the arguments equals *Data*.
device	Finds all IRPs with a stack location where **DeviceObject** equals *Data*.
fileobject	Finds all IRPs whose **Irp.Tail.Overlay.OriginalFileObject** equals *Data*.

CRITERIA	MATCH
mdlprocess	Finds all IRPs whose **Irp.MdlAddress.Process** equals *Data*.
thread	Finds all IRPs whose **Irp.Tail.Overlay.Thread** equals *Data*.
userevent	Finds all IRPs whose **Irp.UserEvent** equals *Data*.

Data
Specifies the data to be matched in the search.

DLL

Kdexts.dll

Additional Information

See Plug and Play Debugging for applications of this extension command. For information about IRPs, see the Windows Driver Kit (WDK) documentation and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

This example finds the IRP in the nonpaged pool that is going to set user event FF9E4F48 upon completion:

```
kd> !irpfind 0 0 userevent ff9e4f48
```

The following example produces a full listing of all IRPs in the nonpaged pool:

```
kd> !irpfind
Searching NonPaged pool (8090c000 : 8131e000) for Tag: Irp
8097c008 Thread 8094d900 current stack belongs to  \Driver\symc810
8097dec8 Thread 8094dda0 current stack belongs to  \FileSystem\Ntfs
809861a8 Thread 8094dda0 current stack belongs to  \Driver\symc810
809864e8 Thread 80951ba0 current stack belongs to  \Driver\Mouclass
80986608 Thread 80951ba0 current stack belongs to  \Driver\Kbdclass
80986728 Thread 8094dda0 current stack belongs to  \Driver\symc810
```

!irpzone

12/5/2018 • 2 minutes to read • Edit Online

The **!irpzone** extension command is obsolete. Use **!irpfind** instead.

!irql

10/23/2019 • 2 minutes to read • Edit Online

The **!irql** extension displays the interrupt request level (IRQL) of a processor on the target computer before the debugger break.

```
!irql [Processor]
```

Parameters

Processor
Specifies the processor. Enter the processor number. If this parameter is omitted, the debugger displays the IRQL of the current processor.

DLL

The **!irql** extension is only available in Windows Server 2003 and later versions of Windows.

Windows 2000	Unavailable
Windows XP	Unavailable
Windows Server 2003 and later	Kdexts.dll

Additional Information

For information about IRQLs, see the Windows Driver Kit (WDK) documentation and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

When the target computer breaks into the debugger, the IRQL changes, but the IRQL that was effective just before the debugger break is saved. The **!irql** extension displays the saved IRQL.

Similarly, when a bug check occurs and a crash dump file is created, the IRQL saved in the crash dump file is the one immediately prior to the bug check, not the IRQL at which the **KeBugCheckEx** routine was executed.

In both cases, the current IRQL is raised to DISPATCH_LEVEL, except on x86 architectures. Thus, if more than one such event occurs, the IRQL displayed will also be DISPATCH_LEVEL, making it useless for debugging purposes.

The **!pcr** extension displays the current IRQL on all versions of Windows, but the current IRQL is usually not useful. The IRQL that existed just before the bug check or debugger connection is more interesting, and this is displayed only with **!irql**.

If you supply an invalid processor number, or there has been kernel corruption, the debugger displays a message "Cannot get PRCB address".

Here is an example of the output from this extension from a dual-processor x86 computer:

```
kd> !irql 0
Debugger saved IRQL for processor 0x0 -- 28 (CLOCK2_LEVEL)

kd> !irql 1
Debugger saved IRQL for processor 0x1 -- 0 (LOW_LEVEL)
```

If the debugger is in verbose mode, a description of the IRQL itself is included. Here is an example from an Itanium processor:

```
kd> !irql
Debugger saved IRQL for processor 0x0 -- 12 (PC_LEVEL) [Performance counter level]
```

The meaning of the IRQL number often depends on the processor. Here is an example from an x64 processor. Note that the IRQL number is the same as in the previous example, but the IRQL meaning is different:

```
kd> !irql
Debugger saved IRQL for processor 0x0 -- 12 (SYNCH_LEVEL) [Synchronization level]
```

!isainfo

12/5/2018 • 2 minutes to read • Edit Online

The **!isainfo** extension displays information about PNPISA cards or devices present in the system..

```
!isainfo [Card]
```

Parameters

Card
Specifies a PNPISA Card. If *Card* is 0 or omitted, all the devices and cards on the PNPISA (that is, the PC I/O) Bus are displayed.

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Kdexts.dll

Remarks

Here is an example of the output from this extension:

```
0: kd> !isainfo
ISA PnP FDO @ 0x867b9938, DevExt @ 0x867b99f0, Bus # 0
Flags (0x80000000)  DF_BUS

  ISA PnP PDO @ 0x867B9818, DevExt @ 0x86595388
  Flags (0x40000818)  DF_ENUMERATED, DF_ACTIVATED,
                      DF_REQ_TRIMMED, DF_READ_DATA_PORT
```

!isr

12/5/2018 • 2 minutes to read • Edit Online

The !isr extension displays the Itanium Interruption Status Register (ISR) at the specified address.

```
!isr Expression [DisplayLevel]
```

Important This command has been deprecated in the Windows Debugger Version 10.0.14257 and later, and is no longer available.

Parameters

Expression
Specifies the hexadecimal address of the ISR register to display. The expression **@isr** can also be used for this parameter. In that case, information about the current processor ISR register is displayed.

DisplayLevel
Can be any one of the following options:

0
Displays only the values of each ISR field. This is the default.

1
Displays detailed information about ISR fields that are not reserved or ignored.

2
Displays details about all ISR fields, including those that are ignored or reserved.

DLL

Windows 2000	Unavailable
Windows XP and later	Kdexts.dll

This extension command can only be used with an Itanium target computer.

Remarks

Here are a couple of examples of output from this extension:

```
kd> !isr @isr
isr:ed ei so ni ir rs sp na r w x vector code
  0  0  0  0  0  0  0  0 0 0 0      0   0

kd> !isr @isr 2

 cod : 0 : interruption Code
 vec : 0 : IA32 exception vector number
  rv : 0 : reserved0
   x : 0 : eXecute exception
   w : 0 : Write exception
   r : 0 : Read exception
  na : 0 : Non-Access exception
  sp : 0 : Speculative load exception
  rs : 0 : Register Stack
  ir : 0 : Invalid Register frame
  ni : 0 : Nested Interruption
  so : 0 : IA32 Supervisor Override
  ei : 0 : Exception IA64 Instruction
  ed : 0 : Exception Deferral
  rv : 0 : reserved1
```

!ivt

12/5/2018 • 2 minutes to read • Edit Online

The !ivt extension displays the Itanium interrupt vector table.

```
!ivt [-v] [-a] [Vector]
!ivt -?
```

Important This command has been deprecated in the Windows Debugger Version 10.0.14257 and later, and is no longer available.

Parameters

Vector
Specifies an interrupt vector table entry for the current processor. If *Vector* is omitted, the entire interrupt vector table for the current processor on the target computer is displayed. Interrupt vectors that have not been assigned are not displayed unless the **-a** option is specified.

-a
Displays all interrupt vectors, including those that are unassigned.

-v
Displays detailed output.

-?
Displays help for this extension in the Debugger Command window.

DLL

Windows 2000	Unavailable
Windows XP and later	Kdexts.dll

This extension command can only be used with an Itanium target computer.

Additional Information

For more information about how to display the interrupt dispatch tables on an x64 or x86 target computer, see **!idt**.

Remarks

Here is an example of the output from this extension:

```
kd> !ivt

Dumping IA64 IVT:

00:e000000083005f60 nt!KiPassiveRelease
0f:e000000083576830 hal!HalpPCIISALine2Pin
10:e0000000830067f0 nt!KiApcInterrupt
20:e000000083006790 nt!KiDispatchInterrupt
30:e000000083576b30 hal!HalpCMCIHandler
31:e000000083576b20 hal!HalpCPEIHandler
41:e000000085039680 i8042prt!I8042KeyboardInterruptService (KINTERRUPT e000000085039620)
51:e000000085039910 i8042prt!I8042MouseInterruptService (KINTERRUPT e0000000850398b0)
61:e0000000854484f0 VIDEOPRT!pVideoPortInterrupt (KINTERRUPT e000000085448490)
71:e0000000856c9450 NDIS!ndisMIsr (KINTERRUPT e0000000856c93f0)
81:e0000000857fd000 SCSIPORT!ScsiPortInterrupt (KINTERRUPT e0000000857fcfa0)
91:e0000000857ff510 atapi!IdePortInterrupt (KINTERRUPT e0000000857ff4b0)
a1:e0000000857d84b0 atapi!IdePortInterrupt (KINTERRUPT e0000000857d8450)
a2:e0000165fff2cab0 portcls!CInterruptSyncServiceRoutine (KINTERRUPT e0000165fff2ca50)
b1:e0000000858c7460 ACPI!ACPIInterruptServiceRoutine (KINTERRUPT e0000000858c7400)
b2:e0000000850382e0 USBPORT!USBPORT_InterruptService (KINTERRUPT e000000085038280)
d0:e0000000835768d0 hal!HalpClockInterrupt
e0:e000000083576850 hal!HalpIpiInterruptHandler
f0:e0000000835769c0 hal!HalpProfileInterrupt
f1:e000000083576830 hal!HalpPCIISALine2Pin
fd:e000000083576b10 hal!HalpMcRzHandler
fe:e000000083576830 hal!HalpPCIISALine2Pin
```

!job

12/5/2018 • 2 minutes to read • Edit Online

The **!job** extension displays a job object.

```
!job [Process [Flags]]
```

Parameters

Process
Specifies the hexadecimal address of a process or a thread whose associated job object is to be examined. If this is omitted, or equal to -1 (in Windows 2000), or equal to zero (in Windows XP and later), the job associated with the current process is displayed.

Flags
Specifies what the display should contain. This can be a sum of any of the following bit values. The default is 0x1:

Bit 0 (0x1)
Causes the display to include job settings and statistics.

Bit 1 (0x2)
Causes the display to include a list of all processes in the job.

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Kdexts.dll

Additional Information

For information about job objects, see *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

Here is an example of the output from this extension:

```
kd> !process 52c
Searching for Process with Cid == 52c
PROCESS 8276c550  SessionId: 0  Cid: 052c    Peb: 7ffdf000  ParentCid: 0060
    DirBase: 01289000  ObjectTable: 825f0368  TableSize:  24.
    Image: cmd.exe
    VadRoot 825609e8 Vads 30 Clone 0 Private 77. Modified 0. Locked 0.
    DeviceMap e1733f38
    Token                             e1681610
    ElapsedTime                       0:00:12.0949
    UserTime                          0:00:00.0359
    .....
    CommitCharge                      109
    Job                               8256e1f0

kd> !job 8256e1f0
Job at ffffffff8256e1f0
  TotalPageFaultCount      0
  TotalProcesses           1
  ActiveProcesses          1
  TotalTerminatedProcesses 0
  LimitFlags               0
  MinimumWorkingSetSize    0
  MaximumWorkingSetSize    0
  ActiveProcessLimit       0
  PriorityClass            0
  UIRestrictionsClass      0
  SecurityLimitFlags       0
  Token                    00000000
```

!kb, !kv

12/5/2018 • 2 minutes to read • Edit Online

The !kb and !kv extension commands are obsolete. Use the **kb (Display Stack Backtrace)** and **kv (Display Stack Backtrace)** commands instead.

!loadermemorylist

12/5/2018 • 2 minutes to read • Edit Online

The **!loadermemorylist** extension displays the memory allocation list that the Windows boot loader passes to Windows.

```
!loadermemorylist ListHeadAddress
```

Parameters

ListHeadAddress
Specifies the address of a list header.

DLL

Windows 2000	Kdexts.dll
Windows XP **Windows Server 2003**	Kdexts.dll
Windows Vista and later	Kdexts.dll

Remarks

This extension is designed to be used at the beginning of the system boot process while Ntldr is running. It displays a memory allocation list that includes the start, end, and type of each page range.

You can stop execution at any point by pressing CTRL+BREAK (in WinDbg) or CTRL+C (in KD).

!lockedpages

12/5/2018 • 2 minutes to read • Edit Online

The **!lockedpages** extension displays driver-locked pages for a specified process.

Syntax

```
!lockedpages [Process]
```

Parameters

Process
Specifies a process. If *Process* is omitted, the current process is used.

DLL

Kdexts.dll

Remarks

You can stop execution at any point by pressing CTRL+BREAK (in WinDbg) or CTRL+C (in KD).

!locks (!kdext*.locks)

7/19/2019 • 2 minutes to read • Edit Online

The **!locks** extension in Kdextx86.dll and Kdexts.dll displays information about kernel ERESOURCE locks.

This extension command should not be confused with the **!ntsdexts.locks** extension command.

```
!locks [Options] [Address]
```

Parameters

Options
Specifies the amount of information to be displayed. Any combination of the following options can be used:

-v
Displays detailed information about each lock.

-p
Display all available information about the locks, including performance statistics.

-d
Display information about all locks. Otherwise, only locks with contention are displayed.)

Address
Specifies the hexadecimal address of the ERESOURCE lock to be displayed. If *Address* is 0 or omitted, information about all ERESOURCE locks in the system will be displayed.

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Kdexts.dll

Remarks

The **!locks** extension displays all locks held on resources by threads. A lock can be shared or exclusive, which means no other threads can gain access to that resource. This information is useful when a deadlock occurs on a system. A deadlock is caused by one non-executing thread holding an exclusive lock on a resource that the executing thread needs.

You can usually pinpoint a deadlock in Microsoft Windows 2000 by finding one non-executing thread that holds an exclusive lock on a resource that is required by an executing thread. Most of the locks are shared.

Here is an example of the basic **!locks** output:

```
kd> !locks
**** DUMP OF ALL RESOURCE OBJECTS ****
KD: Scanning for held locks......

Resource @ 0x80e97620    Shared 4 owning threads
     Threads: ff688da0-01<*> ff687da0-01<*> ff686da0-01<*> ff685da0-01<*>
KD: Scanning for held locks..................................................

Resource @ 0x80e23f38    Shared 1 owning threads
     Threads: 80ed0023-01<*> *** Actual Thread 80ed0020
KD: Scanning for held locks.

Resource @ 0x80d8b0b0    Shared 1 owning threads
     Threads: 80ed0023-01<*> *** Actual Thread 80ed0020
2263 total locks, 3 locks currently held
```

Note that the address for each thread displayed is followed by its thread count (for example, "-01"). If a thread is followed by "<*>", that thread is one of the owners of the lock. In some instances, the initial thread address contains an offset. In that case, the actual thread address is displayed as well.

If you want to find more information about one of these resource objects, use the address that follows "Resource @" as an argument for future commands. To investigate the second resource shown in the preceding example, you could use **dt ERESOURCE 80d8b0b0** or **!thread 80ed0020**. Or you could use the **!locks** extension again with the **-v** option:

```
kd> !locks -v 80d8b0b0

Resource @ 0x80d8b0b0    Shared 1 owning threads
     Threads: 80ed0023-01<*> *** Actual Thread 80ed0020

     THREAD 80ed0020  Cid 4.2c  Teb: 00000000 Win32Thread: 00000000 WAIT: (WrQueue) KernelMode Non-Alertable
         8055e100  Unknown
     Not impersonating
GetUlongFromAddress: unable to read from 00000000
     Owning Process 80ed5238
     WaitTime (ticks)          44294977
     Context Switch Count      147830
     UserTime                  0:00:00.0000
     KernelTime                0:00:02.0143
     Start Address nt!ExpWorkerThread (0x80506aa2)
     Stack Init fafa4000 Current fafa3d18 Base fafa4000 Limit fafa1000 Call 0
     Priority 13 BasePriority 13 PriorityDecrement 0
ChildEBP RetAddr
fafa3d30 804fe997 nt!KiSwapContext+0x25 (FPO: [EBP 0xfafa3d48] [0,0,4]) [D:\NT\base\ntos\ke\i386\ctxswap.asm
@ 139]
fafa3d48 80506a17 nt!KiSwapThread+0x85 (FPO: [Non-Fpo]) (CONV: fastcall) [d:\nt\base\ntos\ke\thredsup.c @
1960]
fafa3d78 80506b36 nt!KeRemoveQueue+0x24c (FPO: [Non-Fpo]) (CONV: stdcall) [d:\nt\base\ntos\ke\queueobj.c @
542]
fafa3dac 805ad8bb nt!ExpWorkerThread+0xc6 (FPO: [Non-Fpo]) (CONV: stdcall) [d:\nt\base\ntos\ex\worker.c @
1130]
fafa3ddc 8050ec72 nt!PspSystemThreadStartup+0x2e (FPO: [Non-Fpo]) (CONV: stdcall)
[d:\nt\base\ntos\ps\create.c @ 2164]
00000000 00000000 nt!KiThreadStartup+0x16 [D:\NT\base\ntos\ke\i386\threadbg.asm @ 81]

1 total locks, 1 locks currently held
```

!logonsession

8/26/2019 • 2 minutes to read • Edit Online

The **!logonsession** extension displays information about a specified logon session.

Free Build Syntax

```
!logonsession LUID
```

Checked Build Syntax

```
!logonsession LUID [InfoLevel]
```

Parameters

LUID
Specifies the locally unique identifier (LUID) of a logon session to display. If *LUID*is 0, information about all logon sessions is displayed.

To display information about the system session and all system tokens in a checked build, enter **!logonsession 3e7 1**.

InfoLevel
(Checked Build Only) Specifies how much token information is displayed. The *InfoLevel* parameter can take values from 0 to 4, with 0 representing the least information and 4 representing the most information.

DLL

Windows 2000	Unavailable
Windows XP and later	Kdexts.dll

Additional Information

For information about logon sessions, see the Microsoft Windows SDK documentation and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

Here is an example of the output from this extension on a free build:

```
kd> !logonsession 0

Dumping all logon sessions.

** Session   0 = 0x0
   LogonId     = {0x0 0x0}
   References  = 0
** Session   1 = 0x8ebb50
 LogonId     = {0xe9f1 0x0}
   References  = 21
** Session   2 = 0x6e31e0
   LogonId     = {0x94d1 0x0}
   References  = 1
** Session   3 = 0x8ecd60
   LogonId     = {0x6b31 0x0}
   References  = 0
** Session   4 = 0xe0000106
   LogonId     = {0x0 0x0}
   References  = 0
** Session   5 = 0x0
   LogonId     = {0x0 0x0}
   References  = 0
** Session   6 = 0x8e9720
   LogonId     = {0x3e4 0x0}
   References  = 6
** Session   7 = 0xe0000106
   LogonId     = {0x0 0x0}
   References  = 0
** Session   8 = 0xa2e160
   LogonId     = {0x3e5 0x0}
   References  = 3
** Session   9 = 0xe0000106
   LogonId     = {0x0 0x0}
   References  = 0
** Session  10 = 0x3ca0
   LogonId     = {0x3e6 0x0}
   References  = 2
** Session  11 = 0xe0000106
   LogonId     = {0x0 0x0}
   References  = 0
** Session  12 = 0x1cd0
   LogonId     = {0x3e7 0x0}
   References  = 33
** Session  13 = 0xe0000106
 LogonId     = {0x0 0x0}
 References  = 0
14 sessions in the system.
```

You can stop execution at any point by pressing CTRL+BREAK (in WinDbg) or CTRL+C (in KD).

!lookaside

10/23/2019 • 2 minutes to read • Edit Online

The **!lookaside** extension displays information about look-aside lists, resets the counters of look-aside lists, or modifies the depth of a look-aside list.

```
!lookaside [Address [Options [Depth]]]
!lookaside [-all]
!lookaside 0 [-all]
```

Parameters

Address
Specifies the hexadecimal address of a look-aside list to be displayed or modified.

If *Address* is omitted (or 0) and the **-all** option is not specified, a set of well-known, standard system look-aside lists is displayed. The set of lists is not exhaustive; that is, it does not include all system look-aside lists. Also, the set does not include custom look-aside lists that were created by calls to **ExInitializePagedLookasideList** or **ExInitializeNPagedLookasideList**.

If *Address* is omitted (or 0) and the **-all** option is specified, all look-aside lists are displayed.

Options
Controls what operation will be taken. The following possible *Options* are supported. The default is zero:

0
Displays information about the specified look-aside list or lists.

1
Resets the counters of the specified look-aside list.

2
Modifies the depth of the specified look-aside list. This option can only be used if *Address* is nonzero.

Depth
Specifies the new maximum depth of the specified look-aside list. This parameter is permitted only if *Address* is nonzero and *Options* is equal to 2.

Additional Information

For information about look-aside lists, see the Windows Driver Kit (WDK) documentation and *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

Look-aside lists are multiprocessor-safe mechanisms for managing pools of fixed-size entries from either paged or nonpaged memory.

Look-aside lists are efficient, because the routines do not use spin locks on most platforms.

Note that if the current depth of a look-aside list exceeds the maximum depth of that list, then freeing a structure associated with that list will result in freeing it into pool memory, rather than list memory.

Here is an example of the output from this extension:

```
!lookaside 0xfffff88001294f80

Lookaside "" @ 0xfffff88001294f80  Tag(hex): 0x7366744e "Ntfs"
    Type           =       0011  PagedPool RaiseIfAllocationFailure
    Current Depth  =          0  Max Depth  =          4
    Size           =        496  Max Alloc  =       1984
    AllocateMisses =          8  FreeMisses =          0
    TotalAllocates =     272492  TotalFrees =     272488
    Hit Rate       =         99% Hit Rate   =        100%
```

Requirements

DLL	Kdexts.dll

!lpc

8/26/2019 • 5 minutes to read • Edit Online

Important Lpc is now emulated in alpc, use the !alpc extension instead.

The **!lpc** extension displays information about all local procedure call (LPC) ports and messages in the target system.

```
!lpc message MessageID
!lpc port Port
!lpc scan Port
!lpc thread Thread
!lpc PoolSearch
!lpc
```

Parameters

message
(Windows Server 2003, Windows XP, and Windows 2000 only) Displays information about a message, such as the server port that contains the message in the queue, and the thread waiting for this message, if any.

MessageID
(Windows Server 2003, Windows XP, and Windows 2000 only) Specifies the message ID of the message to be displayed. If the value of this parameter is 0 or this parameter is omitted, the **!lpc message** command displays a summary list of messages. (In Windows 2000 with Service Pack 1 (SP1), the summary includes all messages in the LPC zone. In Windows 2000 with Service Pack 2 (SP2), Windows XP, and later versions of Windows, the summary includes all messages in the kernel pool. Paged-out messages are not included.)

port
(Windows Server 2003, Windows XP, and Windows 2000 only) Displays port information, such as the name of the port, its semaphore state, messages in its queues, threads in its rundown queue, its handle count, its references, and related ports.

scan
(Windows Server 2003, Windows XP, and Windows 2000 only) Displays summary information about the specified port and about all ports that are connected to it.

Port
(Windows Server 2003, Windows XP, and Windows 2000 only) Specifies the hexadecimal address of the port to be displayed. If the **!lpc port** command is used, and *Port* is 0 or is omitted, a summary list of all LPC ports is displayed. If the **!lpc scan** command is used, *Port* must specify the address of an actual port.

thread
(Windows Server 2003, Windows XP, and Windows 2000 only) Displays port information for all ports that contain the specified thread in their rundown port queues.

Thread
(Windows Server 2003, Windows XP, and Windows 2000 only) Specifies the hexadecimal address of the thread. If this is 0 or omitted, the **!lpc thread** command displays a summary list of all threads that are performing any LPC operations.

PoolSearch

(Windows Server 2003 and Windows XP only) Determines whether the **!lpc message** command searches for messages in the kernel pool. Each time **!lpc PoolSearch** is used, this setting toggles on or off (the initial setting is to not search the kernel pool). This only affects **!lpc message** commands that specify a nonzero value for *MessageID*.

DLL

Windows 2000	Kdextx86.dll
Windows XP **Windows Server 2003**	Kdexts.dll

Additional Information

For information about LPCs, see the Windows Driver Kit (WDK) documentation and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

This extension is not supported in Windows Vista and later versions of Windows.

In Windows Server 2003, Windows XP, and Windows 2000, using **!lpc** with no arguments displays help for this extension in the Debugger Command window.

If you have a thread that is marked as waiting for a reply to a message, use the **!lpc message** command with the ID of the delayed message. This command displays the specified message, the port that contains it, and all related threads.

If the message is not found and there were no read errors (such as "Unable to access zone segment"), the server received the message.

In this case, the server port can usually be found by using the **!lpc thread** command. Threads that are waiting for replies are linked into a server communication queue. This command will display all ports that contain the specified thread. After you know the port address, use the **!lpc port** command. More specific information about each thread can then be obtained by using the **!lpc thread**command with the address of each thread.

Here are several examples of the output from this extension from a Windows XP system:

In this example, all port LPC ports are displayed.

```
kd> !lpc port
Scanning 225 objects
       1  Port: 0xe1405650 Connection: 0xe1405650  Communication: 0x00000000  'SeRmCommandPort'
       1  Port: 0xe141ef50 Connection: 0xe141ef50  Communication: 0x00000000  'SmApiPort'
       1  Port: 0xe13c5740 Connection: 0xe13c5740  Communication: 0x00000000  'ApiPort'
       1  Port: 0xe13d9550 Connection: 0xe13d9550  Communication: 0x00000000  'SbApiPort'
       3  Port: 0xe13d8830 Connection: 0xe141ef50  Communication: 0xe13d8910  '
80000004  Port: 0xe13d8910 Connection: 0xe141ef50  Communication: 0xe13d8830  '
       3  Port: 0xe13d8750 Connection: 0xe13d9550  Communication: 0xe13a4030  '
       .....
```

In the previous example, the port at address e14ae238 has no messages; that is, all messages have been picked up and no new messages have arrived.

```
kd> !lpc port e14ae238

Server connection port e14ae238  Name: ApiPort
 Handles: 1   References: 107
    Server process  : 84aa0140 (csrss.exe)
    Queue semaphore : 84a96da8
 Semaphore state 0 (0x0)
    The message queue is empty
    The LpcDataInfoChainHead queue is empty
```

In the previous example, the port at 0xe14ae238 has messages which have been queued, but not yet picked up by the server.

```
kd> !lpc port 0xe14ae238

Server connection port e14ae238  Name: ApiPort
 Handles: 1   References: 108
    Server process  : 84aa0140 (csrss.exe)
    Queue semaphore : 84a96da8
 Semaphore state 0 (0x0)
        Messages in queue:
 0000 e20d9b80 - Busy  Id=0002249c  From: 0584.0680  Context=00000021  [e14ae248 . e14ae248]
 Length=0098007c  Type=00000001 (LPC_REQUEST)
                  Data: 00000000 0002021e 00000584 00000680 002f0001 00000007
    The message queue contains 1 messages
    The LpcDataInfoChainHead queue is empty
```

The remaining Windows XP examples concern the other options that can be used with this extension.

```
kd> !lpc message 222be
Searching message 222be in threads ...
Client thread 842a4db0 waiting a reply from 222be
Searching thread 842a4db0 in port rundown queues ...

Server communication port 0xe114a3c0
    Handles: 1   References: 1
    The LpcDataInfoChainHead queue is empty
        Connected port: 0xe1e7b948      Server connection port: 0xe14ae238

Client communication port 0xe1e7b948
    Handles: 1   References: 3
    The LpcDataInfoChainHead queue is empty

Server connection port e14ae238  Name: ApiPort
 Handles: 1   References: 107
    Server process  : 84aa0140 (csrss.exe)
    Queue semaphore : 84a96da8
 Semaphore state 0 (0x0)
    The message queue is empty
    The LpcDataInfoChainHead queue is empty
Done.
```

```
kd> !lpc thread 842a4db0
Searching thread 842a4db0 in port rundown queues ...

Server communication port 0xe114a3c0
    Handles: 1   References: 1
    The LpcDataInfoChainHead queue is empty
        Connected port: 0xe1e7b948      Server connection port: 0xe14ae238

Client communication port 0xe1e7b948
    Handles: 1   References: 3
    The LpcDataInfoChainHead queue is empty

Server connection port e14ae238  Name: ApiPort
 Handles: 1   References: 107
    Server process  : 84aa0140 (csrss.exe)
    Queue semaphore : 84a96da8
 Semaphore state 0 (0x0)
    The message queue is empty
    The LpcDataInfoChainHead queue is empty
```

```
kd> !lpc scan e13d8830
Scanning 225 objects
       3  Port: 0xe13d8830 Connection: 0xe141ef50  Communication: 0xe13d8910  '
80000004  Port: 0xe13d8910 Connection: 0xe141ef50  Communication: 0xe13d8830  '
Scanning 3 objects
```

!mca

12/5/2018 • 7 minutes to read • Edit Online

The !mca extension displays the machine check architecture (MCA) registers.

```
!mca
```

Parameters

Address
(Itanium target only) Specifies the address of the MCA error record.

Flags
(Itanium target only) Specifies the level of output. *Flags* can be any combination of the following bits. The default value is 0xFF, which displays all sections present in the log.

Bit 0 (0x1)
Displays the processor section.

Bit 1 (0x2)
Displays the platform-specific section.

Bit 2 (0x4)
Displays the memory section.

Bit 3 (0x8)
Displays the PCI component section.

Bit 4 (0x10)
Displays the PCI bus section.

Bit 5 (0x20)
Displays the SystemEvent Log section.

Bit 6 (0x40)
Displays the platform host controller section.

Bit 7 (0x80)
Displays to include the platform bus section.

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Kdexts.dll

This extension command can only be used with an x86-based or Itanium target computer.

Remarks

On an Itanium target, **!mca** displays the MCA error record from the system abstraction layer (SAL). Here is an

example of the output from this extension:

```
kd> !mca e0000165f3f58000
hal!HalpFeatureBits: 0xf [HAL_PERF_EVENTS|HAL_MCA_PRESENT|HAL_CMC_PRESENT|HAL_CPE_PRESENT]

MCA Error Record Header @ 0xe0000165f3f58000 0xe0000165f3f597a8
   Id               : 8
   Revision         :
      Revision         : 2
      Minor            : 0x2 '
      Major            : 0 '
   ErrorSeverity    : 0 '
   Valid            :
      Valid            : 0 '
      OemPlatformID    : 0y0
      Reserved         : 0y0000000 (0)
   Length           : 0x17a8
   TimeStamp        :
      TimeStamp        : 0x20031106`00134944
      Seconds          : 0x44 'D'
      Minutes          : 0x49 'I'
      Hours            : 0x13 '
      Reserved         : 0 '
      Day              : 0x6 '
      Month            : 0x11 '
      Year             : 0x3 '
      Century          : 0x20 ' '
   OemPlatformId    : [16]  ""


   Severity  : ErrorRecoverable

MCA Error Section Header @ 0xe0000165f3f58028 0xe0000165f3f59578   [Processor]
   Header           :
      Guid             :
         Data1            : 0xe429faf1
         Data2            : 0x3cb7
         Data3            : 0x11d4
         Data4            : [8]  "???"
      Revision         :
         Revision         : 2
         Minor            : 0x2 '
         Major            : 0 '
      RecoveryInfo     :
         RecoveryInfo     : 0 '
         Corrected        : 0y0
         NotContained     : 0y0
 Reset            : 0y0
         Reserved         : 0y0000
         Valid            : 0y0
 Reserved         : 0 '
      Length           : 0x1550
   Valid            :
 Valid            : 0x100101f
      ErrorMap         : 0y1
      StateParameter   : 0y1
      CRLid            : 0y1
      StaticStruct     : 0y1
      CacheCheckNum    : 0y0001
      TlbCheckNum      : 0y0000
      BusCheckNum      : 0y0001
      RegFileCheckNum  : 0y0000
      MsCheckNum       : 0y0000
      CpuIdInfo        : 0y1
      Reserved         : 0y00000000000000000000000000000000000000000000 (0)
   ErrorMap         :
      ErrorMap         : 0x1002000
      Cid              : 0y0000
```

```
      Tid              : 0y0000
      Eic              : 0y0000
      Edc              : 0y0010
      Eit              : 0y0000
      Edt              : 0y0000
      Ebh              : 0y0001
      Erf              : 0y0000
  Ems              : 0y0000000000000000 (0)
      Reserved         : 0y0000000000000000 (0)
    StateParameter   :
      StateParameter   : 0x28000000`fff21130
      reserved0        : 0y00
      rz               : 0y0
  ra               : 0y0
      me               : 0y1
      mn               : 0y1
      sy               : 0y0
      co               : 0y0
      ci               : 0y1
      us               : 0y0
      hd               : 0y0
      tl               : 0y0
      mi               : 0y1
      pi               : 0y0
      pm               : 0y0
      dy               : 0y0
      in               : 0y0
      rs               : 0y1
      cm               : 0y0
      ex               : 0y0
      cr               : 0y1
      pc               : 0y1
      dr               : 0y1
      tr               : 0y1
      rr               : 0y1
      ar               : 0y1
      br               : 0y1
      pr               : 0y1
      fp               : 0y1
      b1               : 0y1
      b0               : 0y1
      gr               : 0y1
      dsize            : 0y0000000000000000 (0)
      reserved1        : 0y00000000000 (0)
      cc               : 0y1
      tc               : 0y0
      bc               : 0y1
      rc               : 0y0
      uc               : 0y0
    CRLid            :
      LocalId          : 0
      reserved         : 0y0000000000000000 (0)
      eid              : 0y00000000 (0)
      id               : 0y00000000 (0)
      ignored          : 0y00000000000000000000000000000000 (0)

    CacheErrorInfo[0]:

    Valid            : 1
    CheckInfo        : 0y1
    RequestorIdentifier : 0y0
    ResponderIdentifier : 0y0
    TargetIdentifier : 0y0
    PreciseIP        : 0y0
  Reserved         : 0y0000000000000000000000000000000000000000000000000000000000000000 (0)
    CheckInfo       : 0x0
    RequestorId     : 0x0
    ResponderId     : 0x0
    TargetIp        : 0x0
```

```
    TargetId        : 0x0
    PreciseIp       : 0x0

    CheckInfo:

    CacheCheck      : 0
    Operation       : 0y0000
    Level           : 0y00
    Reserved1       : 0y00
  DataLine        : 0y0
    TagLine         : 0y0
    DataCache       : 0y0
    InstructionCache : 0y0
    MESI            : 0y000
    MESIValid       : 0y0
    Way             : 0y00000 (0)
    WayIndexValid   : 0y0
    Reserved2       : 0y0000000000 (0)
    Index           : 0y0000000000000000000 (0)
    Reserved3       : 0y00
    InstructionSet  : 0y0
    InstructionSetValid : 0y0
    PrivilegeLevel  : 0y00
    PrivilegeLevelValid : 0y0
    MachineCheckCorrected : 0y0
    TargetAddressValid : 0y0
    RequestIdValid  : 0y0
    ResponderIdValid : 0y0
    PreciseIPValid  : 0y0


    BusErrorInfo[0]:

    Valid           : 9
    CheckInfo       : 0y1
    RequestorIdentifier : 0y0
    ResponderIdentifier : 0y0
    TargetIdentifier : 0y1
    PreciseIP       : 0y0
  Reserved        : 0y00000000000000000000000000000000000000000000000000000000000000 (0)
  CheckInfo       : 0x1080000003000144
    RequestorId     : 0x0
    ResponderId     : 0x0
  TargetIp        : 0x0
    TargetId        : 0xd0022004
    PreciseIp       : 0x0

    CheckInfo:

    BusCheck        : 0x10800000`03000144
    Size            : 0y00100 (0x4)
  Internal        : 0y0
    External        : 0y1
    CacheTransfer   : 0y0
  Type            : 0y00000001 (0x1)
    Severity        : 0y00000 (0)
  Hierarchy       : 0y00
    Reserved1       : 0y0
    Status          : 0y00000011 (0x3)
  Reserved2       : 0y00000000000000000000 (0)
    InstructionSet  : 0y0
    InstructionSetValid : 0y1
  PrivilegeLevel  : 0y00
    PrivilegeLevelValid : 0y0
    MachineCheckCorrected : 0y0
    TargetAddressValid : 0y1
    RequestIdValid  : 0y0
    ResponderIdValid : 0y0
    PreciseIPValid  : 0y0
```

```
  PreciseIPValid   : 0y0

  StaticInfo @ 0xe0000165f3f580f0 0xe0000165f3f59578

  Valid @ 0xe0000165f3f580f0

  Valid            : 0x3f
  MinState         : 0y1
  BR               : 0y1
  CR               : 0y1
  AR               : 0y1
  RR               : 0y1
  FR               : 0y1
  Reserved         : 0y00000000000000000000000000000000000000000000000000000000000000 (0)

MinState @ 0xe0000165f3f580f8 0xe0000165f3f584f0

  IntNats          : 0
IntGp            : 0xe0000165`f1a99b00
IntT0            : 0
  IntT1            : 0xe0f0e0f0`e0f0e000
  IntS0            : 0
IntS1            : 1
  IntS2            : 0xe0000000`83068300
  IntS3            : 0xe0000000`832f8780
  IntV0            : 0x4600
  IntT2            : 0x230
  IntT3            : 0x3ff
IntT4            : 0xe0000165`f38c6000
  IntSp            : 0xe0000165`f0f97da0
  IntTeb           : 0
  IntT5            : 0
  IntT6            : 0xfffff630
  B0R16            : 0x1010`082a6018
  B0R17            : 0
  B0R18            : 0xe0000000`830067c0
  B0R19            : 0x101
  B0R20            : 0x80000000`00000308
  B0R21            : 0
  B0R22            : 0xe0000000`84bedd20
  B0R23            : 0xe0000000`84bedd20
  B0R24            : 0xe0000165`f213df5a
  B0R25            : 0xfff80000`597c84f0
  B0R26            : 0x6081
  B0R27            : 0xfffffe00`00165f20
  B0R28            : 0x8000465
  B0R29            : 0x8000465
  B0R30            : 0x60
  B0R31            : 0xa04`00000000
IntT7            : 0x44
  IntT8            : 0x200
  IntT9            : 0xe0000165`f38c6000
  IntT10           : 0xe0000165`f3cb81bc
  IntT11           : 0xe0000165`f3cb81b8
  IntT12           : 0xe0000000`8363f7b0
  IntT13           : 0xe0000165`f1899d08
  IntT14           : 0x9804c`8a70433f
IntT15           : 0xe0000000`832821f8
  IntT16           : 0xe0000000`836536e0
  IntT17           : 0xe0000000`8363f7b8
  IntT18           : 0xffffffff`fffffbc3
  IntT19           : 0xe0000165`f1ff6000
IntT20           : 0x2400580
  IntT21           : 0xe0000165`f1ff6004
  IntT22           : 0xe0000165`f3cb8dc0
  Preds            : 0x2277
  BrRp             : 0xe0000165`ea212df0
  RsRSC            : 3
  StIIP            : 0xe0000165`f1895370
  StIPSR           : 0x1010`082a6018
```

```
   StIPSR           : 0x1010`082a6018
   StIFS            : 0x80000000`00000285
   XIP              : 0xe0000165`ea212c50
   XPSR             : 0x1010`082a6018
   XFS              : 0x80000000`00000b1c

   BR @ 0xe0000165f3f584f8 0xe0000165f3f58530

e0000165`f3f584f8  e0000165`ea212df0 USBPORT!USBPORT_StopDevice+0x850
e0000165`f3f58500  00000000`00000000
e0000165`f3f58508  00000000`00000000
e0000165`f3f58510  00000000`00000000
e0000165`f3f58518  00000000`00000000
e0000165`f3f58520  00000000`00000000
e0000165`f3f58528  e0000000`832cb061 nt!NtClose+0x1
e0000165`f3f58530  e0000165`f1895320 usbohci!OHCI_StopController

   CR @ 0xe0000165f3f58538 0xe0000165f3f58930

e0000165`f3f58538  00000000`00007e05
e0000165`f3f58540  00000154`a7047201
e0000165`f3f58548  e0000000`83230000 nt!KiVhptTransVector
e0000165`f3f58550  00000000`00000000
...
e0000165`f3f585c8  00000000`00000000
e0000165`f3f585d0  e0000165`f1895370 usbohci!OHCI_StopController+0x50
e0000165`f3f585d8  e0000165`f213df5a
e0000165`f3f585e0  00000000`00000060
e0000165`f3f585e8  e0000165`f1895360 usbohci!OHCI_StopController+0x40
e0000165`f3f585f0  80000000`00000285
...
e0000165`f3f58930  00000000`00000000

   AR @ 0xe0000165f3f58938 0xe0000165f3f58d30

e0000165`f3f58938  00000000`00000000
e0000165`f3f58940  00000000`00000000
e0000165`f3f58948  00000000`00000000
e0000165`f3f58950  00000000`00000000
e0000165`f3f58958  00000000`00000000
e0000165`f3f58960  00000000`00000006
e0000165`f3f58968  e0000000`8301add0 nt!KiMemoryFault
e0000165`f3f58970  00000000`00000000
e0000165`f3f58978  00000000`00000000
e0000165`f3f58980  00000000`00000000
e0000165`f3f58988  00000000`00000000
e0000165`f3f58990  00000000`00000000
e0000165`f3f58998  00000000`00000000
e0000165`f3f589a0  00000000`00000000
e0000165`f3f589a8  00000000`00000000
e0000165`f3f589b0  00000000`00000000
e0000165`f3f589b8  e0000165`f1895370 usbohci!OHCI_StopController+0x50
e0000165`f3f589c0  e0000165`f0f988e0
...
e0000165`f3f58d30  00000000`00000000

   RR @ 0xe0000165f3f58d38 0xe0000165f3f58d70

e0000165`f3f58d38  00000000`00000535
e0000165`f3f58d40  00000000`00000535
e0000165`f3f58d48  00000000`00000535
e0000165`f3f58d50  00000000`00000535
e0000165`f3f58d58  00000000`00000535
e0000165`f3f58d60  00000000`00000535
e0000165`f3f58d68  00000000`00000535
e0000165`f3f58d70  00000000`00000535

   FR @ 0xe0000165f3f58d78 0xe0000165f3f59570
```

```
e0000165`f3f58d78  00000000`00000000
e0000165`f3f58d80  00000000`00000000
e0000165`f3f58d88  80000000`00000000
e0000165`f3f58d90  00000000`0000ffff
e0000165`f3f58d98  00000000`00000000
e0000165`f3f58da0  00000000`00000000
e0000165`f3f58da8  00000000`00000000
e0000165`f3f58db0  00000000`00000000
...
e0000165`f3f59570  00000000`00000000


MCA Error Section Header @ 0xe0000165f3f59578 0xe0000165f3f596a0   [PciComponent]
   Header           :
      Guid             :
         Data1            : 0xe429faf6
         Data2            : 0x3cb7
         Data3            : 0x11d4
         Data4            : [8]  "???"
 Revision         :
         Revision         : 2
         Minor            : 0x2 '
         Major            : 0 '
      RecoveryInfo     :
         RecoveryInfo     : 0x80 '
         Corrected        : 0y0
 NotContained     : 0y0
         Reset            : 0y0
         Reserved         : 0y0000
         Valid            : 0y1
 Reserved         : 0 '
      Length           : 0x128
   Valid            :
 Valid            : 0x23
      ErrorStatus      : 0y1
      Info             : 0y1
 MemoryMappedRegistersPairs : 0y0
      ProgrammedIORegistersPairs : 0y0
      RegistersDataPairs : 0y0
      OemData          : 0y1
      Reserved         : 0y00000000000000000000000000000000000000000000000000000000000000 (0)
   ErrorStatus      :
      Status           : 0x121900
      Reserved0        : 0y00000000 (0)
      Type             : 0y00011001 (0x19)
      Address          : 0y0
 Control          : 0y1
      Data             : 0y0
      Responder        : 0y0
      Requestor        : 0y1
      FirstError       : 0y0
      Overflow         : 0y0
      Reserved1        : 0y00000000000000000000000000000000000000000000000 (0)
   Info             :
      VendorId         : 0x8086
      DeviceId         : 0x100e
 ClassCodeInterface : 0 '
 ClassCodeSubClass : 0 '
      ClassCodeBaseClass : 0x2 '
      FunctionNumber   : 0 '
      DeviceNumber     : 0x3 '
      BusNumber        : 0xa0 '
      SegmentNumber    : 0 '
      Reserved0        : 0 '
      Reserved1        : 0
   MemoryMappedRegistersPairs : 0
   ProgrammedIORegistersPairs : 0

   OemData @ 0xe0000165f3f595b8   0xe0000165f3f596a0
```

```
     Data Length = 0xe6
 Data:
e0000165`f3f595ba  00 00 00 00 00 00 91 d3-86 d3 7a 5e 7e 48 a4 0a  ..........z^~H..
e0000165`f3f595ca  2b f6 f7 a6 cc ca 00 ff-ff ff ff ff ff ff 09 00  +...............
e0000165`f3f595da  00 00 00 00 00 00 00 00-00 00 08 00 00 00 86 80  ................
e0000165`f3f595ea  0e 10 47 01 30 22 08 00-00 00 08 00 00 00 02 00  ..G.0"..........
e0000165`f3f595fa  00 02 20 80 00 00 10 00-00 00 08 00 00 00 00 00  .. .............
e0000165`f3f5960a  00 d0 00 00 00 00 18 00-00 00 08 00 00 00 81 a0  ................
e0000165`f3f5961a  00 00 00 00 00 00 20 00-00 00 08 00 00 00 00 00  ......  .........
e0000165`f3f5962a  00 00 00 00 00 00 28 00-00 00 08 00 00 00 00 00  ......(.........
e0000165`f3f5963a  00 00 3c 10 74 12 30 00-00 00 08 00 00 00 00 00  ..<.t.0.........
e0000165`f3f5964a  00 00 dc 00 00 00 38 00-00 00 08 00 00 00 00 00  ......8.........
e0000165`f3f5965a  00 00 2a 01 ff 00 e4 00-00 00 08 00 00 00 07 f0  ..*.............
e0000165`f3f5966a  1e 00 00 00 40 04 00 00-00 00 00 00 00 00 00 00  ....@...........
e0000165`f3f5967a  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
e0000165`f3f5968a  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
e0000165`f3f5969a  00 00 00 00 00 00                                  ......


MCA Error Section Header @ 0xe0000165f3f596a0 0xe0000165f3f597a8   [PciBus]
   Header           :
 Guid             :
        Data1            : 0xe429faf4
        Data2            : 0x3cb7
        Data3            : 0x11d4
        Data4            : [8]  "???"
 Revision         :
        Revision         : 2
        Minor            : 0x2 '
        Major            : 0 '
     RecoveryInfo     :
        RecoveryInfo     : 0x84 '
        Corrected        : 0y0
 NotContained     : 0y0
        Reset            : 0y1
        Reserved         : 0y0000
        Valid            : 0y1
 Reserved         : 0 '
     Length           : 0x108
  Valid            :
 Valid            : 0x74f
     ErrorStatus      : 0y1
     ErrorType        : 0y1
     Id               : 0y1
     Address          : 0y1
     Data             : 0y0
     CmdType          : 0y0
     RequestorId      : 0y1
     ResponderId      : 0y0
     TargetId         : 0y1
     OemId            : 0y1
 OemData          : 0y1
     Reserved         : 0y00000000000000000000000000000000000000000000000000000000000000 (0)
  ErrorStatus      :
     Status           : 0x121900
     Reserved0        : 0y00000000 (0)
     Type             : 0y00011001 (0x19)
     Address          : 0y0
 Control          : 0y1
     Data             : 0y0
     Responder        : 0y0
     Requestor        : 0y1
     FirstError       : 0y0
     Overflow         : 0y0
 Reserved1        : 0y00000000000000000000000000000000000000000000000000 (0)
  Type             :
     Type             : 0x4 '
     Reserved         : 0 '
```

```
    Id               :
       BusNumber        : 0xa0 '
       SegmentNumber    : 0 '
    Reserved         : [4]  ""
    Address          : 0xd0022054
    Data             : 0
    CmdType          : 0
  RequestorId      : 0xfed2a000
    ResponderId      : 0
    TargetId         : 0xd0022054
    OemId            : [16]  ".???"
    OemData          :
  Length           : 0x98



CP M/R/F/A Manufacturer     SerialNumber     Features         Speed
 0 1,5,31,0 GenuineIntel     0000000000000000 0000000000000001 1000 Mhz
```

On an x86 target, **!mca** displays the machine check registers supported by the active processor. It also displays basic CPU information (identical to that displayed by **!cpuinfo**). Here is an example of the output from this extension:

```
0: kd> !mca
MCE: Enabled, Cycle Address: 0x00000001699f7a00, Type: 0x0000000000000000

MCA: Enabled, Banks 5, Control Reg: Supported, Machine Check: None.
Bank  Error  Control Register     Status Register
  0. None   0x000000000000007f   0x0000000000000000

  1. None   0x00000000ffffffff   0x0000000000000000

  2. None   0x000000000000ffff   0x0000000000000000

  3. None   0x0000000000000007   0x0000000000000000

  4. None   0x0000000000003fff   0x0000000000000000

No register state available.

CP F/M/S Manufacturer   MHz Update Signature Features
 0 15,5,0 SomeBrandName 1394 0000000000000000 a0017fff
```

Note that this extension requires private HAL symbols. Without these symbols, the extension will display the message "HalpFeatureBits not found" along with basic CPU information. For example:

```
kd> !mca
HalpFeatureBits not found
CP F/M/S Manufacturer  MHz Update Signature Features
 0 6,5,1 GenuineIntel  334 0000004000000000 00001fff
```

!memlist

12/5/2018 • 2 minutes to read • Edit Online

The **!memlist** extension scans physical memory lists from the page frame number (PFN) database in order to check them for consistency.

```
!memlist Flags
```

Parameters

Flags
Specifies which memory lists to verify. At present, only one value has been implemented:

Bit 0 (0x1)
Causes the zeroed pages list to be verified.

DLL

Windows 2000	Kdexts.dll
Windows XP and later	Kdexts.dll

Remarks

At present, this extension will only check the zeroed pages list to make sure that all pages in that list are zeroed. The appropriate syntax is:

```
kd> !memlist 1
```

!memusage

4/8/2019 • 3 minutes to read • Edit Online

The **!memusage** extension displays summary statistics about physical memory use.

Syntax

```
!memusage [Flags]
```

Parameters

Flags
Can be any one of the following values. The default is 0x0.

0x0
Displays general summary information, along with a more detailed description of the pages in the PFN database. See the Remarks section for an example of this type of output.

0x1
Displays only summary information about the modified no-write pages in the PFN database..

0x2
Displays only detailed information about the modified no-write pages in the PFN database.

0x8
Displays only general summary information about memory use.

Environment

Modes	kernel mode only

DLL

Kdexts.dll

Additional Information

Physical memory statistics are collected from the Memory Manager's page frame number (PFN) database table.

This command takes a long time to run, especially if the target computer is running in 64-bit mode, due to the greater amount of data to obtain. While it is loading the PFN database, a counter shows its progress. To speed up this loading, increase the COM port speed with the **CTRL+A (Toggle Baud Rate)** key, or use the **.cache (Set Cache Size)** command to increase the cache size (perhaps to around 10 MB).

The **!memusage** command can also be used while performing local kernel debugging.

Here is an example of the output from this extension:

```
kd> !memusage
 loading PFN database
loading (98% complete)

Compiling memory usage data (100% Complete).
             Zeroed:     49 (   196 kb)
               Free:      5 (    20 kb)
            Standby:   5489 ( 21956 kb)
           Modified:    714 (  2856 kb)
    ModifiedNoWrite:      1 (     4 kb)
       Active/Valid:  10119 ( 40476 kb)
         Transition:      6 (    24 kb)
            Unknown:      0 (     0 kb)
              TOTAL:  16383 ( 65532 kb)

  Building kernel map
  Finished building kernel map
Scanning PFN database - (99% complete)

  Usage Summary (in Kb):


Control Valid Standby Dirty Shared Locked PageTables  name

8251a258    12    108     0     0     0     0  mapped_file( cscui.dll )
827ab1b8     8   1708     0     0     0     0  mapped_file( $Mft )
8263c408   908     48     0     0     0     0  mapped_file( win32k.sys )
8252dda8     0    324     0     0     0     0  mapped_file( ShellIconCache )
8272f638   128    112     0   116     0     0  mapped_file( advapi32.dll )
......
82755958     0      4     0     0     0     0  mapped_file( $Directory )
8250b518     0      4     0     0     0     0    No Name for File
8254d8d8     0      4     0     0     0     0  mapped_file( $Directory )
82537be8     0      4     0     0     0     0  mapped_file( Windows Explorer.lnk )

--------  1348      0     0 -----  -----   904  process ( System )
--------   492      0     0 -----  -----    72  process ( winmine.exe )
--------  3364   1384  1396 -----  -----   188  process ( explorer.exe )
--------   972      0     0 -----  -----    88  process ( services.exe )
--------   496   1456   384 -----  -----   164  process ( winmgmt.exe )
--------  1144      0     0 -----  -----   120  process ( svchost.exe )
--------   944      0     0 -----  -----   156  process ( winlogon.exe )
--------   412      0     0 -----  -----    64  process ( csrss.exe )
......
--------    12      0     0 -----  -----     8  process ( wmiadap.exe )

--------   316      0     0 -----  -----     0  pagefile section (346e)
--------  4096      0     0 -----  -----     0  pagefile section (9ad)

--------   884    280    36 -----     0 -----  driver ( ntoskrnl.exe )
--------    88      8     0 -----     0 -----  driver ( hal.dll )
--------     8      0     0 -----     0 -----  driver ( kdcom.dll )
--------    12      0     0 -----     0 -----  driver ( BOOTVID.dll )
......
--------     8      0     0 -----     0 -----  driver ( ndisuio.sys )
--------    16      0     0 -----     0 -----  driver ( dump_scsiport.sys )
--------    56      0     0 -----     0 -----  driver ( dump_aic78xx.sys )
--------  2756   1060   876 -----     0 -----  driver ( Paged Pool )
--------  1936    128   148 -----     0 -----  driver ( Kernel Stacks )
--------     0      0     0 -----     0 -----  driver ( NonPaged Pool )
```

The first column displays the address of the control area structure that describes each mapped structure. Use the **!ca** extension command to display these control areas.

Remarks

You can use the **!vm** extension command to analyze virtual memory use. This extension is typically more useful than **!memusage**. For more information about memory management, see *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

The **!pfn** extension command can be used to display a particular page frame entry in the PFN database.

!mps

12/5/2018 • 2 minutes to read • Edit Online

The **!mps** extension displays BIOS information for the Intel Multiprocessor Specification (MPS) of the target computer.

```
!mps [Address]
```

Parameters

Address
Specifies the hexadecimal address of the MPS table in the BIOS. If this is omitted, the information is obtained from the HAL. This will require HAL symbols.

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Kdexts.dll

This extension command can only be used with an x86-based target computer.

Additional Information

For more information about BIOS debugging, see Debugging BIOS Code. For more information about the MPS, refer to the appropriate version of the Intel MultiProcessor Specification.

!mtrr

12/5/2018 • 2 minutes to read • Edit Online

The **!mtrr** extension displays the contents of the MTRR register.

```
!mtrr
```

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Kdexts.dll

This extension command can only be used with an x86-based target computer.

!npx

12/5/2018 • 2 minutes to read • Edit Online

The **!npx** extension displays the contents of the floating-point register save area.

```
!npx Address
```

Parameters

Address
Specifies the hexadecimal address of the FLOATING_SAVE_AREA structure.

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Kdexts.dll

This extension command can only be used with an x86-based target computer.

!ob, !od, !ow

12/5/2018 • 2 minutes to read • Edit Online

The **!ob**, **!od**, and **!ow** extension commands are obsolete. Use the **ob, od, ow (Output to Port)** commands instead.

!object

12/5/2018 • 3 minutes to read • Edit Online

The **!object** extension displays information about a system object.

```
!object Address [Flags]
!object Path
!object 0 Name
!object -p
!object {-h|-?}
```

Parameters

Address
If the first argument is a nonzero hexadecimal number, it specifies the hexadecimal address of the system object to be displayed.

Flags
Specifies the level of detail in the command output.

Set *Flags* to a bitwise OR of these values:

0x0
Display object type.

0x1
Display object type, object name, and reference counts.

0x8
Display the contents of an object directory or the target of a symbolic link. This flag has an effect only if **0x1** is also set.

0x10
Display optional object headers.

0x20
Display the full path to a named object. This flag has an effect only if **0x1** is also set.

The *Flags* parameter is optional. The default value is 0x9.

Path
If the first argument begins with a backslash (), **!object** interprets it as an object path name. When this option is used, the display will be arranged according to the directory structure used by the Object Manager.

Name
If the first argument is zero, the second argument is interpreted as the name of a class of system objects for which to display all instances.

-p
Display private object name spaces.

{**-h**|**-?**}
Display help for this command.

DLL

Kdexts.dll

Examples

This example passes the path of the \Device directory to **!object**. The output lists all objects in the \Device directory.

```
0: kd> !object \Device
Object: ffffc00b074166a0  Type: (ffffe0083b768690) Directory
    ObjectHeader: ffffc00b07416670 (new version)
    HandleCount: 0  PointerCount: 224
    Directory Object: ffffc00b074092e0  Name: Device

    Hash Address          Type          Name
    ---- -------          ----          ----
     00  ffffe0083e6a61f0 Device        00000044
         ffffe0083dcc4050 Device        00000030
         ffffe0083d34f050 Device        NDMP2
         ffffe0083bdf7060 Device        NTPNP_PCI0002
         ...
         ffffe0083b85d060 Device        USBPDO-8
         ffffe0083d33d050 Device        USBFDO-6
         ...
         ffffe0083bdf0060 Device        NTPNP_PCI0001
```

Choose one of listed objects, say USBPDO-8. Pass the address of USBPDO-8 (ffffe0083b85d060) to **!objec**t. Set *Flags* to 0x0 to get minimal information.

```
0: kd> !object ffffe0083b85d060 0x0
Object: ffffe0083b85d060  Type: (ffffe0083b87df20) Device
    ObjectHeader: ffffe0083b85d030 (new version)
```

Include name and reference count information for the same object by setting *Flags* to 0x1.

```
0: kd> !object ffffe0083b85d060 0x1
Object: ffffe0083b85d060  Type: (ffffe0083b87df20) Device
    ObjectHeader: ffffe0083b85d030 (new version)
    HandleCount: 0  PointerCount: 6
    Directory Object: ffffc00b074166a0  Name: USBPDO-8
```

Get optional header information for the same object by setting *Flags* to 0x10.

```
0: kd> !object ffffe0083b85d060 0x10
Object: ffffe0083b85d060  Type: (ffffe0083b87df20) Device
    ObjectHeader: ffffe0083b85d030 (new version)
Optional Headers:
    NameInfo(ffffe0083b85d010)
```

The following example calls **!object** twice for a Directory object. The first time, the contents of the directory are not displayed because the 0x8 flag is not set. The second time, the contents of the directory are displayed because both the 0x8 and 0x1 flags are set (Flags = 0x9).

```
0: kd> !object ffffc00b07481d00 0x1
Object: ffffc00b07481d00  Type: (ffffe0083b768690) Directory
    ObjectHeader: ffffc00b07481cd0 (new version)
    HandleCount: 0  PointerCount: 3
    Directory Object: ffffc00b07481eb0  Name: Filters

0: kd> !object ffffc00b07481d00 0x9
Object: ffffc00b07481d00  Type: (ffffe0083b768690) Directory
    ObjectHeader: ffffc00b07481cd0 (new version)
    HandleCount: 0  PointerCount: 3
    Directory Object: ffffc00b07481eb0  Name: Filters

    Hash Address          Type          Name
    ---- -------          ----          ----
     19  ffffe0083c5f56e0 Device        FltMgrMsg
     21  ffffe0083c5f5060 Device        FltMgr
```

The following example calls **!object** twice for a SymbolicLink object. The first time, the target of the symbolic link is not displayed because the 0x8 flag is not set. The second time, the target of the symbolic link is splayed because both the 0x8 and 0x1 flags are set (Flags = 0x9).

```
0: kd> !object ffffc00b07628fb0 0x1
Object: ffffc00b07628fb0  Type: (ffffe0083b769450) SymbolicLink
    ObjectHeader: ffffc00b07628f80 (new version)
    HandleCount: 0  PointerCount: 1
    Directory Object: ffffc00b074166a0  Name: Ip6

0: kd> !object ffffc00b07628fb0 0x9
Object: ffffc00b07628fb0  Type: (ffffe0083b769450) SymbolicLink
    ObjectHeader: ffffc00b07628f80 (new version)
    HandleCount: 0  PointerCount: 1
    Directory Object: ffffc00b074166a0  Name: Ip6
    Target String is '\Device\Tdx'
```

Additional Information

For information about objects and the object manager, see the Microsoft Windows SDK documentation, the Windows Driver Kit (WDK) documentation, and *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

See also

Object Reference Tracing

!obtrace

!handle

Determining the ACL of an Object

Kernel-Mode Extension Commands

!obtrace

12/5/2018 • 2 minutes to read • Edit Online

The **!obtrace** extension displays object reference tracing data for the specified object.

```
!obtrace Object
```

Parameters

Object
A pointer to the object or a path.

DLL

Windows 2000	Unavailable
Windows XP and later	Kdexts.dll

Additional Information

For more information about the Global Flags utility (GFlags), see the Windows Driver Kit (WDK) documentation and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

The object reference tracing feature of Windows records sequential stack traces whenever an object reference counter is incremented or decremented.

Before using this extension to display object reference tracing data, you must use GFlags to enable object reference tracing for the specified object. You can enable object reference tracing as a kernel flag (run-time) setting, in which the change is effective immediately, but disappears if you shut down or restart; or as a registry setting, which requires a restart, but remains effective until you change it.

Here is an example of the output from the **!obtrace** extension:

```
kd> !obtrace 0xfa96f700
Object: fa96f700        Image: cmd.exe
Sequence  (+/-)  Stack
--------  -----  --------------------------------------------------
   2421d    +1  nt!ObCreateObject+180
                nt!NtCreateEvent+92
                nt!KiFastCallEntry+104
                nt!ZwCreateEvent+11
                win32k!UserThreadCallout+6f
                win32k!W32pThreadCallout+38
                nt!PsConvertToGuiThread+174
                nt!KiBBTUnexpectedRange+c

   2421e    -1  nt!ObfDereferenceObject+19
                nt!NtCreateEvent+d4
                nt!KiFastCallEntry+104
                nt!ZwCreateEvent+11
                win32k!UserThreadCallout+6f
                win32k!W32pThreadCallout+38
                nt!PsConvertToGuiThread+174
                nt!KiBBTUnexpectedRange+c

   2421f    +1  nt!ObReferenceObjectByHandle+1c3
                win32k!xxxCreateThreadInfo+37d
                win32k!UserThreadCallout+6f
                win32k!W32pThreadCallout+38
                nt!PsConvertToGuiThread+174
                nt!KiBBTUnexpectedRange+c

   24220    +1  nt!ObReferenceObjectByHandle+1c3
                win32k!ProtectHandle+22
                win32k!xxxCreateThreadInfo+3a0
                win32k!UserThreadCallout+6f
                win32k!W32pThreadCallout+38
                nt!PsConvertToGuiThread+174
                nt!KiBBTUnexpectedRange+c

   24221    -1  nt!ObfDereferenceObject+19
                win32k!xxxCreateThreadInfo+3a0
                win32k!UserThreadCallout+6f
                win32k!W32pThreadCallout+38
                nt!PsConvertToGuiThread+174
                nt!KiBBTUnexpectedRange+c

----  ------------------------------------------------------------
References: 3, Dereferences 2
```

The primary indicators in the **!obtrace 0xfa96f700** display are listed in the following table.

PARAMETER	MEANING
Sequence	Represents the order of operations.
+/-	Indicates a reference or a dereference operation. +1 indicates a reference operation. -1 indicates a dereference operation. +/- n indicates multiple reference/dereference operations.

The object reference traces on x64-based target computers might be incomplete because it is not always possible to acquire stack traces at IRQL levels higher than PASSIVE_LEVEL.

You can stop execution at any time by pressing CTRL+BREAK (in WinDbg) or CTRL+C (in KD).

!openmaps

8/26/2019 • 2 minutes to read • Edit Online

The **!openmaps** extension displays the referenced buffer control blocks (BCBs) and virtual address control blocks (VACBs) for the specified shared cache map.

```
!openmaps Address [Flag]
```

Parameters

Address
Specifies the address of the shared cache map.

Flag
Determines which control blocks are displayed. If *Flag* is **1**, the debugger displays all control blocks. If *Flag* is **0**, the debugger displays only referenced control blocks. The default is **0**.

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Kdexts.dll

Additional Information

For information about cache management, see the Microsoft Windows SDK documentation and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

For information about other cache management extensions, see the **!cchelp** extension.

!pars

12/5/2018 • 2 minutes to read • Edit Online

The **!pars** extension displays a specified processor application registers file.

```
!pars Address
```

Important This command has been deprecated in the Windows Debugger Version 10.0.14257 and later, and is no longer available.

Parameters

Address
Specifies the address of a processor application registers file.

DLL

Windows 2000	Unavailable
Windows XP and later	Kdexts.dll

This extension command can only be used with an Itanium-based target computer.

!pat

12/5/2018 • 2 minutes to read • Edit Online

The **!pat** extension displays the Page Attribute Table (PAT) registers for the target processor.

```
!pat Flag
!pat
```

Parameters

Flag
If *Flag* is set, the debugger verifies that the PAT feature is present before the PAT is displayed.

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Kdexts.dll

This extension command can only be used with an x86-based target computer.

!pci

12/5/2018 • 5 minutes to read • Edit Online

The **!pci** extension displays the current status of the peripheral component interconnect (PCI) buses, as well as any devices attached to those buses.

```
!pci [Flags [Segment] [Bus [Device [Function [MinAddress MaxAddress]]]]]
```

Parameters

Flags
Specifies the level of output. Can be any combination of the following bits:

Bit 0 (0x1)
Causes a verbose display.

Bit 1 (0x2)
Causes the display to include all buses in the range from bus 0 (zero) to the specified *Bus*.

Bit 2 (0x4)
Causes the display to include information in raw byte format. If *MinAddress*, *MaxAddress*, or flag bit 0x8 is set, this bit is automatically set as well.

Bit 3 (0x8)
Causes the display to include information in raw DWORD format.

Bit 4 (0x10)
Causes the display to include invalid device numbers. If *Device* is specified, this flag is ignored.

Bit 5 (0x20)
Causes the display to include invalid function numbers.

Bit 6 (0x40)
Causes the display to include capabilities.

Bit 7 (0x80)
Causes the display to include Intel 8086 device-specific information.

Bit 8 (0x100)
Causes the display to include the PCI configuration space.

Bit 9 (0x200)
Causes the display to include segment information. When this bit is included, the *Segment* parameter must be included.

Bit 10 (0x400)
Causes the display to include all valid segments in the range from segment 0 to the specified segment. When this bit is included, the *Segment* parameter must be included.

Segment
Specifies the number of the segment to be displayed. Segment numbers range from 0 to 0xFFFF. If *Segment* is omitted, information about the primary segment (segment 0) is displayed. If *Flags* includes bit 10 (0x400), *Segment* specifies the highest valid segment to be displayed.

Bus
Specifies the bus to be displayed. *Bus* can range from 0 to 0xFF. If it is omitted, information about the primary bus (bus 0) is displayed. If *Flags* includes bit 1 (0x2), *Bus* specifies the highest bus number to be displayed.

Device
Specifies the slot device number for the device. If this is omitted, information about all devices is printed.

Function
Specifies the slot function number for the device. If this is omitted, all information about all device functions is printed.

MinAddress
Specifies the first address from which raw bytes or DWORDs are to be displayed. This must be between 0 and 0xFF.

MaxAddress
Specifies the last address from which raw bytes or DWORDs are to be displayed. This must be between 0 and 0xFF, and not less than *MinAddress*.

DLL

Windows 2000	Kext.dll Kdextx86.dll
Windows XP and later	Kext.dll

This extension command can only be used with an x86-based target computer.

Additional Information

See Plug and Play Debugging for applications of this extension command and additional examples. For information about PCI buses, see the Windows Driver Kit (WDK) documentation.

Remarks

To edit the PCI configuration space, use **!ecb**, **!ecd**, or **!ecw**.

The following example displays a list of all buses and their devices. This command will take a long time to execute. You will see a moving counter at the bottom of the display while the debugger scans the target system for PCI buses:

```
kd> !pci 2 ff
PCI Bus 0
00:0  8086:1237.02  Cmd[0106:.mb..s]  Sts[2280:.....]  Device  Host bridge
0d:0  8086:7000.01  Cmd[0007:imb...]  Sts[0280:.....]  Device  ISA bridge
0d:1  8086:7010.00  Cmd[0005:i.b...]  Sts[0280:.....]  Device  IDE controller
0e:0  1011:0021.02  Cmd[0107:imb..s]  Sts[0280:.....]  PciBridge 0->1-1  PCI-PCI bridge
10:0  102b:0519.01  Cmd[0083:im....]  Sts[0280:.....]  Device  VGA compatible controller
PCI Bus 1
08:0  10b7:9050.00  Cmd[0107:imb..s]  Sts[0200:.....]  Device  Ethernet
09:0  9004:8178.00  Cmd[0117:imb..s]  Sts[0280:.....]  Device  SCSI controller
```

This example displays verbose information about the devices on the primary bus. The two-digit number at the beginning of each line is the device number; the one-digit number following it is the function number:

```
kd> !pci 1 0
PCI Bus 0
00:0  8086:1237.02  Cmd[0106:.mb..s]  Sts[2280:.....]  Device  Host bridge
      cf8:80000000  IntPin:0  IntLine:0  Rom:0  cis:0  cap:0

0d:0  8086:7000.01  Cmd[0007:imb...]  Sts[0280:.....]  Device  ISA bridge
      cf8:80006800  IntPin:0  IntLine:0  Rom:0  cis:0  cap:0

0d:1  8086:7010.00  Cmd[0005:i.b...]  Sts[0280:.....]  Device  IDE controller
      cf8:80006900  IntPin:0  IntLine:0  Rom:0  cis:0  cap:0
      IO[4]:fff1

0e:0  1011:0021.02  Cmd[0107:imb..s]  Sts[0280:.....]  PciBridge 0->1-1  PCI-PCI bridge
      cf8:80007000  IntPin:0  IntLine:0  Rom:0  cap:0  2sts:2280  BCtrl:6 ISA
      IO:f000-ffff  Mem:fc000000-fdffffff  PMem:fff00000-fffff

10:0  102b:0519.01  Cmd[0083:im....]  Sts[0280:.....]  Device  VGA compatible controller
      cf8:80008000  IntPin:1  IntLine:9  Rom:80000000  cis:0  cap:0
      MEM[0]:fe800000  MPF[1]:fe000008
```

This example shows even more detailed information about bus 0 (zero), device 0x0D, and function 0x1, including the raw DWORDS from addresses between 0x00 and 0x3F:

```
kd> !pci f 0 d 1 0 3f
PCI Bus 0
0d:1  8086:7010.00  Cmd[0005:i.b...]  Sts[0280:.....]  Device  IDE controller
      cf8:80006900  IntPin:0  IntLine:0  Rom:0  cis:0  cap:0
      IO[4]:fff1
      00000000:  70108086 02800005 01018000 00002000
      00000010:  00000000 00000000 00000000 00000000
      00000020:  0000fff1 00000000 00000000 00000000
      00000030:  00000000 00000000 00000000 00000000
```

This example displays the configuration space for segment 1, bus 0, device 1:

```
0: kd> !pci 301 1 0 1

PCI Configuration Space (Segment:0001 Bus:00 Device:01 Function:00)
Common Header:
    00: VendorID       14e4 Broadcom Corporation
    02: DeviceID       16c7
    04: Command        0146 MemSpaceEn BusInitiate PERREn SERREn
    06: Status         02b0 CapList 66MHzCapable FB2BCapable DEVSELTiming:1
.
.
.
    5a: MsgCtrl        64BitCapable MultipleMsgEnable:0 (0x1) MultipleMsgCapable:3 (0x8)
    5c: MsgAddr        2d4bff00
    60: MsgAddrHi      1ae09097
    64: MsData         9891
```

To display all devices and buses on valid segments, issue the command **!pci 602 ffff ff**:

```
0: kd> !pci 602 ffff ff
Scanning the following PCI segments: 0 0x1
PCI Segment 0 Bus 0
01:0  14e4:16c7.10  Cmd[0146:.mb.ps]  Sts[02b0:c6...]  Ethernet Controller  SubID:103c:1321
02:0  1000:0030.08  Cmd[0147:imb.ps]  Sts[0230:c6...]  LSI SCSI Controller  SubID:103c:1323
02:1  1000:0030.08  Cmd[0147:imb.ps]  Sts[0230:c6...]  LSI SCSI Controller  SubID:103c:1323
03:0  1000:0030.08  Cmd[0147:imb.ps]  Sts[0230:c6...]  LSI SCSI Controller  SubID:103c:1323
03:1  1000:0030.08  Cmd[0147:imb.ps]  Sts[0230:c6...]  LSI SCSI Controller  SubID:103c:1323
PCI Segment 0 Bus 0x38
01:0  14e4:1644.12  Cmd[0146:.mb.ps]  Sts[02b0:c6...]  Ethernet Controller  SubID:10b7:1000
PCI Segment 0 Bus 0x54
00:0  103c:403b.00  Cmd[0547:imb.ps]  Sts[0010:c....]  HP PCI-PCI Bridge 0x54->0x55-0x55
PCI Segment 0 Bus 0x70
00:0  103c:403b.00  Cmd[0547:imb.ps]  Sts[0010:c....]  HP PCI-PCI Bridge 0x70->0x71-0x71
PCI Segment 0 Bus 0xa9
01:0  8086:b154.00  Cmd[0147:imb.ps]  Sts[0ab0:c6.A.]  Intel PCI-PCI Bridge 0xa9->0xaa-0xaa
PCI Segment 0 Bus 0xaa
04:0  1033:0035.41  Cmd[0146:.mb.ps]  Sts[0210:c....]  NEC USB Controller  SubID:103c:1293
04:1  1033:0035.41  Cmd[0146:.mb.ps]  Sts[0210:c....]  NEC USB Controller  SubID:103c:aa55
04:2  1033:00e0.02  Cmd[0146:.mb.ps]  Sts[0210:c....]  NEC USB2 Controller  SubID:103c:aa55
05:0  1002:5159.00  Cmd[0187:imb..s]  Sts[0290:c....]  ATI VGA Compatible Controller  SubID:103c:1292
PCI Segment 0 Bus 0xc6
00:0  103c:403b.00  Cmd[0547:imb.ps]  Sts[0010:c....]  HP PCI-PCI Bridge 0xc6->0xc7-0xc7
PCI Segment 0 Bus 0xe3
00:0  103c:403b.00  Cmd[0547:imb.ps]  Sts[0010:c....]  HP PCI-PCI Bridge 0xe3->0xe4-0xe4
PCI Segment 0x1 Bus 0
01:0  14e4:16c7.10  Cmd[0146:.mb.ps]  Sts[02b0:c6...]  Ethernet Controller  SubID:103c:1321
02:0  1000:0030.08  Cmd[0147:imb.ps]  Sts[0230:c6...]  LSI SCSI Controller  SubID:103c:1323
02:1  1000:0030.08  Cmd[0147:imb.ps]  Sts[0230:c6...]  LSI SCSI Controller  SubID:103c:1323
03:0  1000:0030.08  Cmd[0147:imb.ps]  Sts[0230:c6...]  LSI SCSI Controller  SubID:103c:1323
03:1  1000:0030.08  Cmd[0147:imb.ps]  Sts[0230:c6...]  LSI SCSI Controller  SubID:103c:1323
PCI Segment 0x1 Bus 0x54
00:0  103c:403b.00  Cmd[0547:imb.ps]  Sts[0010:c....]  HP PCI-PCI Bridge 0x54->0x55-0x55
PCI Segment 0x1 Bus 0x55
00:0  8086:10b9.06  Cmd[0147:imb.ps]  Sts[0010:c....]  Intel Ethernet Controller  SubID:8086:1083
PCI Segment 0x1 Bus 0x70
00:0  103c:403b.00  Cmd[0547:imb.ps]  Sts[0010:c....]  HP PCI-PCI Bridge 0x70->0x71-0x71
PCI Segment 0x1 Bus 0xc6
00:0  103c:403b.00  Cmd[0547:imb.ps]  Sts[0010:c....]  HP PCI-PCI Bridge 0xc6->0xc7-0xc7
PCI Segment 0x1 Bus 0xe3
00:0  103c:403b.00  Cmd[0547:imb.ps]  Sts[0010:c....]  HP PCI-PCI Bridge 0xe3->0xe4-0xe4
```

!pciir

12/5/2018 • 2 minutes to read • Edit Online

The **!pciir** extension displays the contents of the hardware routing of peripheral component interconnect (PCI) devices to interrupt controller inputs.

```
!pciir
```

DLL

Windows 2000	Kdextx86.dll
Windows XP **Windows Server 2003**	Kdexts.dll
Windows Vista and later	Unavailable

This extension command can only be used with an x86-based target computer that does not have the Advanced Configuration and Power Interface (ACPI) enabled.

Additional Information

For similar information on any ACPI-enabled computer, use the **!acpiirqarb** extension.

For information about PCI buses, see the Windows Driver Kit (WDK) documentation.

!pcitree

12/5/2018 • 2 minutes to read • Edit Online

The **!pcitree** extension displays information about PCI device objects, including child PCI buses and CardBus buses, and the devices attached to them.

```
!pcitree
```

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Kdexts.dll

Additional Information

See Plug and Play Debugging for applications of this extension command. For information about PCI buses and PCI device objects, see the Windows Driver Kit (WDK) documentation.

Remarks

Here is an example:

```
kd> !pcitree

Bus 0x0 (FDO Ext fe517338)
  0600 12378086 (d=0,  f=0) devext fe4f4ee8 Bridge/HOST to PCI
  0601 70008086 (d=d,  f=0) devext fe4f4ce8 Bridge/PCI to ISA
  0101 70108086 (d=d,  f=1) devext fe4f4ae8 Mass Storage Controller/IDE
  0604 00211011 (d=e,  f=0) devext fe4f4788 Bridge/PCI to PCI

Bus 0x1 (FDO Ext fe516998)
  0200 905010b7 (d=8,  f=0) devext fe515ee8 Network Controller/Ethernet
  0100 81789004 (d=9,  f=0) devext fe515ce8 Mass Storage Controller/SCSI
  0300 0519102b (d=10, f=0) devext fe4f4428 Display Controller/VGA

Total PCI Root busses processed = 1
```

To understand this display, consider the final device shown. Its base class is 03, its subclass is 00, its Device ID is 0x0519, and its Vendor ID is 0x102B. These values are all intrinsic to the device itself.

The number after "d=" is the device number; the number after "f=" is the function number. After "devext" is the device extension address, 0xFE4F4428. Finally, the base class name and the subclass name appear.

To obtain more information about a device, use the **!devext** extension command with the device extension address as the argument. For this particular device, the command to use would be:

```
kd> !devext fe4f4428 pci
```

If the **!pcitree** extension generates an error, this often means that your PCI symbols were not loaded properly. Use **.reload pci.sys** to fix this problem.

!pcm

8/26/2019 • 2 minutes to read • Edit Online

The **!pcm** extension displays the specified private cache map. This extension is only available in Windows 2000.

```
!pcm Address
```

Parameters

Address
Specifies the address of the private cache map.

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Unavailable (see Remarks section)

Additional Information

For information about cache management, see the Microsoft Windows SDK documentation and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

For information about other cache management extensions, see the **!cchelp** extension reference.

Remarks

This extension is supported only in Windows 2000. In Windows XP and later versions of Windows, use the **dt nt!_PRIVATE_CACHE_MAP Address** command.

!pcr

10/8/2019 • 2 minutes to read • Edit Online

The **!pcr** extension displays the current status of the Processor Control Region (PCR) on a specific processor.

```
!pcr [Processor]
```

Parameters

Processor
Specifies the processor to retrieve the PCR information from. If *Processor* is omitted, the current processor is used.

> **NOTE**
> This command is not currently supported and may display incorrect output.

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Kdexts.dll

Additional Information

For information about the PCR and the PRCB, see *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

The processor control block (PRCB) is an extension of the PCR. It can be displayed with the **!prcb** extension.

Here is an example of the **!pcr** extension on an x86 target computer:

```
kd> !pcr 0
KPCR for Processor 0 at ffdff000:
    Major 1 Minor 1
      NtTib.ExceptionList: 801626e0
          NtTib.StackBase: 801628f0
         NtTib.StackLimit: 8015fb00
       NtTib.SubSystemTib: 00000000
            NtTib.Version: 00000000
        NtTib.UserPointer: 00000000
            NtTib.SelfTib: 00000000

                  SelfPcr: ffdff000
                     Prcb: ffdff120
                     Irql: 00000000
                      IRR: 00000000
                      IDR: ffffffff
            InterruptMode: 00000000
                      IDT: 80043400
                      GDT: 80043000
                      TSS: 803cc000

            CurrentThread: 8015e8a0
               NextThread: 00000000
               IdleThread: 8015e8a0

                DpcQueue:  0x80168ee0 0x80100d04 ntoskrnl!KiTimerExpiration
```

One of the entries in this display shows the interrupt request level (IRQL). The **!pcr** extension shows the current IRQL, but the current IRQL is usually not of much interest. The IRQL that existed just before the bug check or debugger connection is more interesting. This is displayed by **!irql**, which is only available on computers running Windows Server 2003 or later versions of Windows.

!pcrs

12/5/2018 • 2 minutes to read • Edit Online

The **!pcrs** extension displays the Intel Itanium-specific processor control registers.

```
!pcrs Address
```

Important This command has been deprecated in the Windows Debugger Version 10.0.14257 and later, and is no longer available.

Parameters

Address
Specifies the address of a processor control registers file.

DLL

Windows 2000	Unavailable
Windows XP and later	Kdexts.dll

This extension command can only be used with an Itanium-based target computer.

Remarks

Do not confuse the **!pcrs** extension with the **!pcr** extension, which displays the current status of the processor control region.

!pfn

4/8/2019 • 2 minutes to read • Edit Online

The **!pfn** extension displays information about a specific page frame or the entire page frame database.

```
!pfn PageFrame
```

Parameters

PageFrame
Specifies the hexadecimal number of the page frame to be displayed.

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Kdexts.dll

Additional Information

For information about page tables, page directories, and page frames, see *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

The page frame number for a virtual address can be obtained by using the **!pte** extension.

Here is an example of the output from this extension:

```
kd> !pte 801544f4
801544F4  - PDE at C0300800        PTE at C0200550
          contains 0003B163      contains 00154121
        pfn 3b G-DA--KWV    pfn 154 G--A--KRV

kd> !pfn 3b
    PFN 0000003B at address 82350588
    flink       00000000  blink / share count 00000221  pteaddress C0300800
    reference count 0001                                color 0
    restore pte 00000000  containing page        000039  Active


kd> !pfn 154
    PFN 00000154 at address 82351FE0
    flink       00000000  blink / share count 00000001  pteaddress C0200550
    reference count 0001                                color 0
    restore pte 00000060  containing page        00003B  Active     M
    Modified
```

!pmc

12/5/2018 • 2 minutes to read • Edit Online

The **!pmc** extension displays the Performance Monitor Counter (PMC) register at the specified address.

This extension is supported only on an Itanium-based target computer.

```
!pmc [- Option] Expression [DisplayLevel]
```

Important This command has been deprecated in the Windows Debugger Version 10.0.14257 and later, and is no longer available.

Parameters

Option
Can be any one of the following values:

gen
Displays the register as a generic PMC register.

btb
Displays the register as a branch trace buffer (BTB) PMC register.

Expression
Specifies the hexadecimal address of a PMC. The expressions **@kpfcgen** and **@kpfcbtb** can be used as values for this parameter.

If *Expression* is **@kpfcgen**, the debugger displays the current processor PMC register as a generic PMC register. You can also display the current processor PMC register as a generic PMC register by setting *Option* to **gen** and using **@kpfc4**, **@kpfc5**, **@kpfc6**, or **@kpfc7** for the *Expression* value.

If *Expression* is **@kpfcbtb**, the debugger displays the current processor PMC register as a BTB PMC register. You can also display the current processor PMC register as a BTB PMC register by setting *Option* to **btb** and using @kpfc12 for the *Expression* value.

DisplayLevel
Can be any one of the following values:

0
Displays only the values of each PMC register field. This is the default.

1
Displays detailed information about the PMC register fields that are not reserved or ignored.

2
Displays detailed information about all PMC register fields, including those that are ignored or reserved.

DLL

Windows 2000	Unavailable

Windows XP and later	Kdexts.dll

!pmssa

12/5/2018 • 2 minutes to read • Edit Online

The **!pmssa** extension displays a specified processor Minimal State Save Area (also known as Min-StateSave Area).

This extension can only be used with an Itanium-based target computer.

```
!pmssa Address
```

Important This command has been deprecated in the Windows Debugger Version 10.0.14257 and later, and is no longer available.

Parameters

Address
Specifies the address of a processor Min-StateSave Area.

DLL

Windows 2000	Unavailable
Windows XP and later	Kdexts.dll

!powertriage

6/25/2019 • 2 minutes to read • Edit Online

The !powertriage extension displays summary information about the system and device power related components. It also provides links to related commands that can be used to gather additional information. The !powertriage command has no parameters. This command can be used with both live kernel-mode debugging and for crash dump file analysis.

Syntax

```
!powertriage
```

Parameters

None

DLL

Windows 10 and later	Kdexts.dll

Remarks

The !powertriage extension displays the following information.

1. Power state of the device node along with !podev for all the device objects.
2. Links to **!rcdrkd.rcdrlogdump** if the driver has enabled the IFR. For more information about IFR, see Using Inflight Trace Recorder (IFR) in KMDF and UMDF 2 Drivers.
3. Links to **!wdfkd.wdfdriverinfo** and **!wdfkd.wdflogdump** for WDF drivers.
4. Links to !fxdevice for PoFx devices. For more information about PoFX, see Overview of the Power Management Framework. Here is example output from the !powertriage command.

```
kd> !powertriage

System Capabilities :
  Machine is not AOAC capable.

Power Capabilities:
PopCapabilities @ 0xfffff8022f6c4380
  Misc Supported Features:  PwrButton S1 S3 S4 S5 HiberFile FullWake
  Processor Features:
  Disk Features:
  Battery Features:
  Wake Caps
    Ac OnLine Wake:         Sx
    Soft Lid Wake:          Sx
    RTC Wake:               S4
    Min Device Wake:        Sx
    Default Wake:           Sx



Power Action:
```

```
PopAction :fffff8022f6ba550
    Current System State..: Working
    Target System State...: Unspecified
    State.................: - Idle(0)

Devices with allocated Power IRPs:

    +  ACPI\PNP0C0C\2&daba3ff&1
       0xffffe00023939ad0 ACPI D0 !podev  WAIT_WAKE_IRP !irp Related Threads

    +  USB\ROOT_HUB30\5&2c60645a&0&0
       0xffffe0002440ac40 USBXHCI D2 !podev  WAIT_WAKE_IRP !irp Related Threads !rcdrlogdump !wdfdriverinfo
!wdflogdump
         Upper DO 0xffffe00024415a10 USBHUB3 !podev


    +  USB\ROOT_HUB30\5&d91dce5&0&0
       0xffffe00023ed4d30 USBXHCI D2 !podev  WAIT_WAKE_IRP !irp Related Threads !rcdrlogdump !wdfdriverinfo
!wdflogdump
         Upper DO 0xffffe000249d8040 USBHUB3 !podev

    +  PCI\VEN_8086&DEV_27E2&SUBSYS_01DE1028&REV_01\3&172e68dd&0&E5
       0xffffe000239e5880 pci D0 !podev FxDevice: !fxdevice  WAIT_WAKE_IRP !irp Related Threads
         Upper DO 0xffffe000239c0e50 ACPI !podev
           Upper DO 0xffffe000239f7040 pci !podev


    +  PCI\VEN_14E4&DEV_167A&SUBSYS_01DE1028&REV_02\4&24ac2e11&0&00E5
       0xffffe000231e6060 pci D0 !podev  WAIT_WAKE_IRP !irp Related Threads
         Upper DO 0xffffe00024359050 b57nd60a !podev


Device Tree Info:

    !devpowerstate

    !devpowerstate Complete


Links:
!poaction
!cstriage
!pdctriage
!pdcclients
!fxdevice
!pnptriage
```

Dump File Power Failure Analysis

The !powertriage extension can be useful in examining system crashes related to incorrect power state information. For example, in the case of **Bug Check 0x9F: DRIVER_POWER_STATE_FAILURE**, the extension will display all the allocated power IRPs, the associated device stacks along with:

1. Links to the **!irp** command for the related IRPs.
2. Links to the **!findthreads** command with the related IRP. The IRP is added as part of the search criteria and displays the threads starting with higher correlation to the search criteria threads listed first. Dumping all device stacks with power IRPs can help in debugging cases where !analyze has not been able to correctly identify the IRP associated with the crash.

!pnpevent

12/5/2018 • 2 minutes to read • Edit Online

The **!pnpevent** extension displays the Plug and Play device event queue.

```
!pnpevent [DeviceEvent]
```

Parameters

DeviceEvent
Specifies the address of a device event to display. If this is zero or omitted, the tree of all device events in the queue is displayed.

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Kdexts.dll

Additional Information

See Plug and Play Debugging for applications of this extension command. For information about Plug and Play drivers, see the Windows Driver Kit (WDK) documentation.

See also

Plug and Play and Power Debugger Commands

!pocaps

8/26/2019 • 2 minutes to read • Edit Online

The **!pocaps** extension displays the power capabilities of the target computer.

```
!pocaps
```

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Kdexts.dll

Additional Information

To view the system's power policy, use the **!popolicy** extension command. For information about power capabilities and power policy, see the Windows Driver Kit (WDK) documentation and *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

Here is an example of this command's output:

```
kd> !pocaps
PopCapabilities @ 0x8016b100
  Misc Supported Features:  S4 FullWake
  Processor Features:
  Disk Features:            SpinDown
  Battery Features:
  Wake Caps
    Ac OnLine Wake:         Sx
    Soft Lid Wake:          Sx
    RTC Wake:               Sx
    Min Device Wake:        Sx
    Default Wake:           Sx
```

!pool

8/26/2019 • 2 minutes to read • Edit Online

The **!pool** extension displays information about a specific pool allocation or about the entire system-wide pool.

```
!pool [Address [Flags]]
```

Parameters

Address
Specifies the pool entry to be displayed. If *Address* is -1, this command displays information about all heaps in the process.

If *Address* is 0 or omitted, this command displays information about the process heap.

Flags
Specifies the level of detail to be used. This can be any combination of the following bit values; the default is zero:

Bit 0 (0x1)
Causes the display to include the pool contents, not just the pool headers.

Bit 1 (0x2)
Causes the display to suppress pool header information for all pools, except the one that actually contains the specified *Address*.

Bit 31 (0x80000000)
Suppresses the description of the pool type and pool tag in the display.

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Kdexts.dll

Additional Information

For information about memory pools, see the Windows Driver Kit (WDK) documentation and *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

In Windows XP and later versions of Windows, the **!pool** extension displays the pool tag associated with each allocation. The owner of that pool tag is also displayed. This display is based on the contents of the pooltag.txt file. This file is located in the triage subdirectory of your Debugging Tools for Windows installation. If you want , you can edit this file to add additional pool tags relevant to your project.

Warning If you install an updated version of Debugging Tools for Windows in the same directory as the current version, it overwrites all of the files in that directory, including pooltag.txt. If you modify or replace the sample pooltag.txt file, be sure to save a copy of it to a different directory. After reinstalling the debuggers, you can copy the saved pooltag.txt over the default version.

If the **!pool** extension reports pool corruption, you should use **!poolval** to investigate.

Here is an example. If *Address* specifies 0xE1001050, the headers of all pools in this block are displayed, and 0xE1001050 itself is marked with an asterisk (*).

```
kd> !pool e1001050
 e1001000 size:   40 previous size:    0  (Allocated)  MmDT
 e1001040 size:   10 previous size:   40  (Free)       Mm
*e1001050 size:   10 previous size:   10  (Allocated) *ObDi
 e1001060 size:   10 previous size:   10  (Allocated)  ObDi
 e1001070 size:   10 previous size:   10  (Allocated)  Symt
 e1001080 size:   40 previous size:   10  (Allocated)  ObDm
 e10010c0 size:   10 previous size:   40  (Allocated)  ObDi
.....
```

In this example, the right-most column shows the pool tag. The column to the left of this shows whether the pool is free or allocated.

The following command shows the pool headers and pool contents:

```
kd> !pool e1001050 1
 e1001000 size:   40 previous size:    0  (Allocated)  MmDT
 e1001008  ffffffff 0057005c 004e0049 004f0044
    e1001018  ffffffff 0053005c 00730079 00650074

 e1001040 size:   10 previous size:   40  (Free)       Mm
 e1001048  ffffffff e1007ba8 e1501a58 01028101
    e1001058  ffffffff 00000000 e1000240 01028101

*e1001050 size:   10 previous size:   10  (Allocated) *ObDi
 e1001058  ffffffff 00000000 e1000240 01028101
    e1001068  ffffffff 00000000 e10009c0 01028101

 e1001060 size:   10 previous size:   10  (Allocated)  ObDi
 e1001068  ffffffff 00000000 e10009c0 01028101
    e1001078  ffffffff 00000000 00000000 04028101

......
```

!poolfind

7/9/2019 • 2 minutes to read • Edit Online

The **!poolfind** extension finds all instances of a specific pool tag in either nonpaged or paged memory pools.

```
!poolfind TagString [PoolType]
!poolfind TagValue [PoolType]
```

Parameters

TagString
Specifies the pool tag. *TagString* is a case-sensitive ASCII string. The asterisk (*) can be used to represent any number of characters; the question mark (?) can be used to represent exactly one character. Unless an asterisk is used, *TagString* must be exactly four characters in length.

TagValue
Specifies the pool tag. *TagValue* must begin with "0x", even if the default radix is 16. If this parameter begins with any other value (including "0X") it will be interpreted as an ASCII tag string.

PoolType
Specifies the type of pool to be searched. The following values are permitted:

0
Specifies nonpaged memory pool. This is the default.

1
Specifies paged memory pool.

2
Specifies the special pool.

4
Specifies the session pool.

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Kdexts.dll

Additional Information

For information about memory pools and pool tags, see the Windows Driver Kit (WDK) documentation and *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

This command can take a significant amount of time to execute, depending on the size of pool memory that must be searched. To speed up this execution, increase the COM port speed with the **CTRL+A (Toggle Baud Rate)** key, or use the **.cache (Set Cache Size)** command to increase the cache size (to approximately 10 MB).

The pool tag is the same tag passed to the **ExAllocate***Xxx* family of routines.

Here is an example. The entire nonpaged pool is searched and then the paged pool is searched, but the command is terminated before completion (after an hour of operation):

```
kd> !poolfind SeSd 0

Scanning large pool allocation table for Tag: SeSd (827d1000 : 827e9000)

Searching NonPaged pool (823b1000 : 82800000) for Tag: SeSd

826fa130 size:   c0 previous size:   40  (Allocated) SeSd
82712000 size:   c0 previous size:    0  (Allocated) SeSd
82715940 size:   a0 previous size:   60  (Allocated) SeSd
8271da30 size:   c0 previous size:   10  (Allocated) SeSd
82721c00 size:   10 previous size:   30  (Free)      SeSd
8272b3f0 size:   60 previous size:   30  (Allocated) SeSd
8272d770 size:   60 previous size:   40  (Allocated) SeSd
8272d7d0 size:   a0 previous size:   60  (Allocated) SeSd
8272d960 size:   a0 previous size:   70  (Allocated) SeSd
82736f30 size:   a0 previous size:   10  (Allocated) SeSd
82763840 size:   a0 previous size:   10  (Allocated) SeSd
8278b730 size:  100 previous size:  290  (Allocated) SeSd
8278b830 size:   10 previous size:  100  (Free)      SeSd
82790130 size:   a0 previous size:   20  (Allocated) SeSd
82799180 size:   a0 previous size:   10  (Allocated) SeSd
827c00e0 size:   a0 previous size:   30  (Allocated) SeSd
827c8320 size:   a0 previous size:   60  (Allocated) SeSd
827ca180 size:   a0 previous size:   50  (Allocated) SeSd
827ec140 size:   a0 previous size:   10  (Allocated) SeSd

Searching NonPaged pool (fe7c3000 : ffbe0000) for Tag: SeSd

kd> !poolfind SeSd 1

Scanning large pool allocation table for Tag: SeSd (827d1000 : 827e9000)

Searching Paged pool (e1000000 : e4400000) for Tag: SeSd

e10000b0 size:   d0 previous size:   20  (Allocated) SeSd
e1000260 size:   d0 previous size:   60  (Allocated) SeSd
......
e1221dc0 size:   a0 previous size:   60  (Allocated) SeSd
e1224250 size:   a0 previous size:   30  (Allocated) SeSd

...terminating - searched pool to e1224000
kd>
```

!poolused

8/26/2019 • 3 minutes to read • Edit Online

The **!poolused** extension displays memory use summaries, based on the tag used for each pool allocation.

```
!poolused [Flags [TagString]]
```

Parameters

Flags
Specifies the amount of output to be displayed and the method of sorting the output. This can be any combination of the following bit values, except that bits 1 (0x2) and 2 (0x4) cannot be used together. The default is 0x0, which produces summary information, sorted by pool tag.

Bit 0 (0x1)
Displays more detailed (verbose) information.

Bit 1 (0x2)
Sorts the display by the amount of nonpaged memory use.

Bit 2 (0x4)
Sorts the display by the amount of paged memory use.

Bit 3 (0x8)
(Windows Server 2003 and later) Displays the session pool instead of the standard pool.

Note You can use the **!session** command to switch between sessions.

TagString
Specifies the pool tag. *TagString* is a case-sensitive ASCII string. The asterisk (*) can be used to represent any number of characters; the question mark (?) can be used to represent exactly one character. Unless an asterisk is used, *TagString* must be exactly four characters in length.

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Kdexts.dll

Additional Information

For information about memory pools and pool tags, see the Windows Driver Kit (WDK) documentation and *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

The **!poolused** extension gathers data from the pool tagging feature of Windows. Pool tagging is permanently enabled on Windows Server 2003 and later versions of Windows. On Windows XP and earlier versions of Windows, you must enable pool tagging by using Gflags.

If you stop executing the extension before it completes, the debugger displays partial results.

The display for this command shows the memory use for each tag in paged pool and nonpaged pool. In both cases, the display includes the number of currently outstanding allocations for the given tag and the number of bytes being consumed by those allocations.

Here is a partial example of the output from this extension:

```
0: kd> !poolused
   Sorting by  Tag

  Pool Used:
            NonPaged            Paged
 Tag    Allocs     Used    Allocs     Used
 1394        1      520         0        0UNKNOWN pooltag '1394', please update pooltag.txt
 1MEM        1     3368         0        0UNKNOWN pooltag '1MEM', please update pooltag.txt
 2MEM        1     3944         0        0UNKNOWN pooltag '2MEM', please update pooltag.txt
 3MEM        3      248         0        0UNKNOWN pooltag '3MEM', please update pooltag.txt
 8042        4     3944         0        0PS/2 kb and mouse , Binary: i8042prt.sys
 AGP         1      344         2      384UNKNOWN pooltag 'AGP ', please update pooltag.txt
 AcdN        2     1072         0        0TDI AcdObjectInfoG
 AcpA        3      192         1      504ACPI Pooltags , Binary: acpi.sys
 AcpB        0        0         4      576ACPI Pooltags , Binary: acpi.sys
 AcpD       40    13280         0        0ACPI Pooltags , Binary: acpi.sys
 AcpF        6      240         0        0ACPI Pooltags , Binary: acpi.sys
 AcpM        0        0         1      128ACPI Pooltags , Binary: acpi.sys
 AcpO        4      208         0        0ACPI Pooltags , Binary: acpi.sys

...

 WmiG       30     6960         0        0Allocation of WMIGUID
 WmiR       63     4032         0        0Wmi Registration info blocks
 Wmip      146     3504       182    18600Wmi General purpose allocation
 Wmit        1     4096         7    49480Wmi Trace
 Wrpa        2      720         0        0WAN_ADAPTER_TAG
 Wrpc        1       72         0        0WAN_CONN_TAG
 Wrpi        1      120         0        0WAN_INTERFACE_TAG
 Wrps        2      128         0        0WAN_STRING_TAG
 aEoP        1      672         0        0UNKNOWN pooltag 'aEoP', please update pooltag.txt
 fEoP        1       16         0        0UNKNOWN pooltag 'fEoP', please update pooltag.txt
 hSVD        0        0         1       40Shared Heap Tag , Binary: mrxdav.sys
 hibr        0        0         1    24576UNKNOWN pooltag 'hibr', please update pooltag.txt
 iEoP        1       24         0        0UNKNOWN pooltag 'iEoP', please update pooltag.txt
 idle        2      208         0        0Power Manager idle handler
 jEoP        1       24         0        0UNKNOWN pooltag 'jEoP', please update pooltag.txt
 mEoP        1       88         0        0UNKNOWN pooltag 'mEoP', please update pooltag.txt
 ohci        1      136         0        01394 OHCI host controller driver
 rx..        3     1248         0        0UNKNOWN pooltag '  rx', please update pooltag.txt
 sidg        2       48         0        0GDI spooler events
 thdd        0        0         1    20480DirectDraw/3D handle manager table
 usbp       18    77056         2       96UNKNOWN pooltag 'usbp', please update pooltag.txt
 vPrt        0        0        18    68160UNKNOWN pooltag 'vPrt', please update pooltag.txt
 TOTAL     3570214 209120008     38769 13066104
```

!poolval

8/26/2019 • 2 minutes to read • Edit Online

The **!poolval** extension analyzes the headers for a pool page and diagnoses any possible corruption. This extension is only available in Windows XP and later versions.

```
!poolval Address [DisplayLevel]
```

Parameters

Address
Specifies the address of the pool whose header is to be analyzed.

DisplayLevel
Specifies the information to include in the display. This can be any of the following values (the default is zero):

0
Causes basic information to be displayed.

1
Causes basic information and linked header lists to be displayed.

2
Causes basic information, linked header lists, and basic header information to be displayed.

3
Causes basic information, linked header lists, and full header information to be displayed.

DLL

Windows 2000	Unavailable
Windows XP and later	Kdexts.dll

Additional Information

For information about memory pools, see the Windows Driver Kit (WDK) documentation and *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

!popolicy

8/26/2019 • 2 minutes to read • Edit Online

The **!popolicy** extension displays the power policy of the target computer.

```
!popolicy [Address]
```

Parameters

Address
Specifies the address of the power policy structure to display. If this is omitted, then nt!PopPolicy is displayed.

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Kdexts.dll

Additional Information

To view the system's power capabilities, use the **!pocaps** extension command. For information about power capabilities and power policy, see the Windows Driver Kit (WDK) documentation and *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

Here is an example of this command's output:

```
kd> !popolicy
SYSTEM_POWER_POLICY (R.1) @ 0x80164d58
  PowerButton:      Shutdown  Flags: 00000003   Event: 00000000   Query UI
  SleepButton:          None  Flags: 00000003   Event: 00000000   Query UI
  LidClose:             None  Flags: 00000001   Event: 00000000   Query
  Idle:                 None  Flags: 00000001   Event: 00000000   Query
  OverThrottled:        None  Flags: c0000004   Event: 00000000   Override NoWakes Critical
  IdleTimeout:             0  IdleSensitivity:        50%
  MinSleep:               S0  MaxSleep:               S0
  LidOpenWake:            S0  FastSleep:              S0
  WinLogonFlags:           1  S4Timeout:               0
  VideoTimeout:            0  VideoDim:              209
  SpinTimeout:             0  OptForPower:             1
  FanTolerance:            0% ForcedThrottle:          0%
  MinThrottle:             0%
```

See also

Plug and Play and Power Debugger Commands

!pplookaside

6/25/2019 • 2 minutes to read • Edit Online

The **!pplookaside** command displays Lookaside Lists for processors in the target computer.

```
!pplookaside <address>
```

Parameters

ParamName
The address of the processor.

DLL

!ppmidle

12/5/2018 • 2 minutes to read • Edit Online

The **!ppmidle** command displays processor idle states.

```
!ppmidle
```

Parameters

None

This command is supported on the following versions of Windows:

- Windows 7
- Windows 8
- Windows 8.1
- Windows 10, Version 1511
- Windows 10, Version 1607
- Windows 10, Version 1703

DLL

Kdexts.dll

!ppmidleaccounting

12/5/2018 • 2 minutes to read • Edit Online

The **!ppmidleaccounting** command displays processor idle state accounting information.

```
!ppmidleaccounting
```

Parameters

None

This command is supported on the following versions of Windows:

- Windows 7
- Windows 8
- Windows 8.1
- Windows 10, Version 1511
- Windows 10, Version 1607
- Windows 10, Version 1703

DLL

Kdexts.dll

!ppmidlepolicy

12/5/2018 • 2 minutes to read • Edit Online

The **!ppmidlepolicy** command displays C-state policy.

```
!ppmidlepolicy
```

Parameters

None

This command is supported on the following versions of Windows:

- Windows 7

DLL

Kdexts.dll

!ppmlpscenarios

12/5/2018 • 2 minutes to read • Edit Online

The **!ppmlpscenarios** command displays per-scenario policy overrides.

```
!ppmlpscenarios
```

Parameters

None

This command is supported on the following versions of Windows:

- Windows 8

DLL

Kdexts.dll

!ppmperf

12/5/2018 • 2 minutes to read • Edit Online

The **!ppmperf** command displays power and performance constraint information.

```
!ppmperf
```

Parameters

None

DLL

kedexts.dll

NOTE: This command is not supported on all versions of Windows, such as Windows 10, Version 1703.

!ppmperfpolicy

12/5/2018 • 2 minutes to read • Edit Online

The **!ppmperfpolicy** command is no longer supported, use the !ppmsettings command instead.

```
!ppmperfpolicy
```

Parameters

None

DLL

Kdexts.dll

!ppmstate

12/5/2018 • 2 minutes to read • Edit Online

The **!ppmstate** command displays power state information.

```
!ppmstate
```

Parameters

None

DLL

Kdexts.dll

NOTE: This command is not supported on all versions of Windows, such as Windows 10, Version 1703.

!ppmstate

12/5/2018 • 2 minutes to read • Edit Online

The **!ppmstate** command displays power state information.

```
!ppmstate
```

Parameters

None

DLL

Kdexts.dll

NOTE: This command is not supported on all versions of Windows, such as Windows 10, Version 1703.

!prcb

8/26/2019 • 2 minutes to read • Edit Online

The **!prcb** extension displays the processor control block (PRCB).

```
!prcb [Processor]
```

Parameters

Processor
Specifies the processor to retrieve the PRCB information from. If *Processor* is omitted, processor zero will be used.

DLL

Windows 2000	Unavailable
Windows XP and later	Kdexts.dll

Additional Information

For information about the PCR and the PRCB, see *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

The PRCB is an extension of the processor control region (PCR). To display the PCR, use the **!pcr** extension.

Here is an example:

```
kd> !prcb
PRCB for Processor 0 at e0000000818ba000:
Threads--  Current e0000000818bbe10 Next 0000000000000000 Idle e0000000818bbe10
Number 0 SetMember 00000001
Interrupt Count -- 0000b81f
Times -- Dpc    00000028 Interrupt 000003ff
         Kernel 00005ef4 User      00000385
```

!process

12/5/2018 • 7 minutes to read • Edit Online

The !process extension displays information about the specified process, or about all processes, including the EPROCESS block.

This extension can be used only during kernel-mode debugging.

Syntax

```
!process [/s Session] [/m Module] [Process [Flags]]
!process [/s Session] [/m Module] 0 Flags ImageName
```

Parameters

/s **** *Session*
Specifies the session that owns the desired process.

/m **** *Module*
Specifies the module that owns the desired process.

Process
Specifies the hexadecimal address or the process ID of the process on the target computer.

The value of *Process* determines whether the !process extension displays a process address or a process ID . If *Process* is omitted in any version of Windows, the debugger displays data only about the current system process. If *Process* is 0 and *ImageName* is omitted, the debugger displays information about all active processes. If -1 is specified for *Process* information about the current process is displayed.

Flags
Specifies the level of detail to display. *Flags* can be any combination of the following bits. If *Flags* is 0, only a minimal amount of information is displayed. The default varies according to the version of Windows and the value of *Process*. The default is 0x3 if *Process* is omitted or if *Process* is either 0 or -1; otherwise, the default is 0xF.

Bit 0 (0x1)
Displays time and priority statistics.

Bit 1 (0x2)
Displays a list of threads and events associated with the process, and their wait states.

Bit 2 (0x4)
Displays a list of threads associated with the process. If this is included without Bit 1 (0x2), each thread is displayed on a single line. If this is included along with Bit 1, each thread is displayed with a stack trace.

Bit 3 (0x8)
Displays the return address and the stack pointer for each function The display of function arguments is suppressed.

Bit 4 (0x10)
Sets the process context equal to the specified process for the duration of this command. This results in a more accurate display of thread stacks. Because this flag is equivalent to using **.process /p /r** for the specified process, any existing user-mode module list will be discarded. If *Process* is zero, the debugger

displays all processes, and the process context is changed for each one. If you are only displaying a single process and its user-mode state has already been refreshed (for example, with **.process /p /r**), it is not necessary to use this flag. This flag is only effective when used with Bit 0 (0x1).

ImageName
Specifies the name of the process to be displayed. The debugger displays all processes whose executable image names match *ImageName*. The image name must match that in the EPROCESS block. In general, this is the executable name that was invoked to start the process, including the file extension (usually .exe), and truncated after the fifteenth character. There is no way to specify an image name that contains a space. When *ImageName* is specified, *Process* must be zero.

DLL

Kdexts.dll

Additional Information

For information about processes in kernel mode, see Changing Contexts. For more information about analyzing processes and threads, see *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

The following is an example of a **!process 0 0** display:

```
kd> !process 0 0
**** NT ACTIVE PROCESS DUMP ****
PROCESS 80a02a60  Cid: 0002    Peb: 00000000  ParentCid: 0000
    DirBase: 00006e05  ObjectTable: 80a03788  TableSize: 150.
    Image: System
PROCESS 80986f40  Cid: 0012    Peb: 7ffde000  ParentCid: 0002
    DirBase: 000bd605  ObjectTable: 8098fce8  TableSize:  38.
    Image: smss.exe
PROCESS 80958020  Cid: 001a    Peb: 7ffde000  ParentCid: 0012
    DirBase: 0008b205  ObjectTable: 809782a8  TableSize: 150.
    Image: csrss.exe
PROCESS 80955040  Cid: 0020    Peb: 7ffde000  ParentCid: 0012
    DirBase: 00112005  ObjectTable: 80955ce8  TableSize:  54.
    Image: winlogon.exe
PROCESS 8094fce0  Cid: 0026    Peb: 7ffde000  ParentCid: 0020
    DirBase: 00055005  ObjectTable: 80950cc8  TableSize: 222.
    Image: services.exe
PROCESS 8094c020  Cid: 0029    Peb: 7ffde000  ParentCid: 0020
    DirBase: 000c4605  ObjectTable: 80990fe8  TableSize: 110.
    Image: lsass.exe
PROCESS 809258e0  Cid: 0044    Peb: 7ffde000  ParentCid: 0026
    DirBase: 001e5405  ObjectTable: 80925c68  TableSize:  70.
    Image: SPOOLSS.EXE
```

The following table describes some of the elements of the **!process 0 0** output.

ELEMENT	MEANING
Process address	The eight-character hexadecimal number after the word PROCESS is the address of the EPROCESS block. In the final entry in the preceding example, the process address is 0x809258E0.

ELEMENT	MEANING
Process ID (PID)	The hexadecimal number after the word Cid. In the final entry in the preceding example, the PID is 0x44, or decimal 68.
Process Environment Block (PEB)	The hexadecimal number after the word Peb is the address of the process environment block. In the final entry in the preceding example, the PEB is located at address 0x7FFDE000.
Parent process PID	The hexadecimal number after the word ParentCid is the PID of the parent process. In the final entry in the preceding example, the parent process PID is 0x26, or decimal 38.
Image	The name of the module that owns the process. In the final entry in the preceding example, the owner is spoolss.exe. In the first entry, the owner is the operating system itself.
Process object address	The hexadecimal number after the word ObjectTable. In the final entry in the preceding example, the address of the process object is 0x80925c68.

To display full details on one process, set *Flags* to 7. The process itself can be specified by setting *Process* equal to the process address, setting *Process* equal to the process ID, or setting *ImageName* equal to the executable image name. Here is an example:

```
kd> !process fb667a00 7
PROCESS fb667a00 Cid: 0002  Peb: 00000000 ParentCid: 0000
  DirBase: 00030000 ObjectTable: e1000f88 TableSize: 112.
  Image: System
  VadRoot fb666388 Clone 0 Private 4. Modified 9850. Locked 0.
  FB667BBC MutantState Signalled OwningThread 0
  Token                e10008f0
  ElapsedTime            15:06:36.0338
  UserTime             0:00:00.0000
  KernelTime            0:00:54.0818
  QuotaPoolUsage[PagedPool]     1480
Working Set Sizes (now,min,max) (3, 50, 345)
  PeakWorkingSetSize        118
  VirtualSize            1 Mb
  PeakVirtualSize          1 Mb
  PageFaultCount          992
  MemoryPriority          BACKGROUND
  BasePriority           8
  CommitCharge           8

    THREAD fb667780 Cid 2.1 Teb: 00000000 Win32Thread: 80144900 WAIT: (WrFreePage) KernelMode Non-
Alertable
    80144fc0 SynchronizationEvent
    Not impersonating
    Owning Process fb667a00
    WaitTime (seconds)   32278
    Context Switch Count  787
    UserTime         0:00:00.0000
    KernelTime        0:00:21.0821
    Start Address Phase1Initialization (0x801aab44)
    Initial Sp fb26f000 Current Sp fb26ed00
    Priority 0 BasePriority 0 PriorityDecrement 0 DecrementCount 0

    ChildEBP RetAddr Args to Child
    fb26ed18 80118efc c0502000 804044b0 00000000 KiSwapThread+0xb5
    fb26ed3c 801289d9 80144fc0 00000008 00000000 KeWaitForSingleObject+0x1c2
```

Note that the address of the process object can be used as input to other extensions, such as **!handle**, to obtain further information.

The following table describes some of the elements in the previous example.

ELEMENT	MEANING
WAIT	The parenthetical comment after this heading gives the reason for the wait. The command **dt nt!_KWAIT_REASON** will display a list of all wait reasons.
ElapsedTime	Lists the amount of time that has elapsed since the process was created. This is displayed in units of Hours:Minutes:Seconds.Milliseconds.
UserTime	Lists the amount of time the process has been running in user mode. If the value for UserTime is exceptionally high, it might identify a process that is depleting system resources. Units are the same as those of ElapsedTime.

ELEMENT	MEANING
KernelTime	Lists the amount of time the process has been running in kernel mode. If the value for KernelTime is exceptionally high, it might identify a process that is depleting system resources. Units are the same as those of ElapsedTime.
Working Set sizes	Lists the current, minimum and maximum working set size for the process, in pages. An exceptionally large working set size can be a sign of a process that is leaking memory or depleting system resources.
QuotaPoolUsage entries	Lists the paged and nonpaged pool used by the process. On a system with a memory leak, looking for excessive nonpaged pool usage on all the processes can tell you which process has the memory leak.
Clone	Indicates whether or not the process was created by the POSIX or Interix subsystems.
Private	Indicates the number of private (non-sharable) pages currently being used by the process. This includes both paged in and paged out memory.

In addition to the process list information, the thread information contains a list of the resources on which the thread has locks. This information is listed in the third line of output after the thread header. In this example, the thread has a lock on one resource, a SynchronizationEvent with an address of 80144fc0. By comparing this address to the list of locks shown by the **!kdext*.locks** extension, you can determine which threads have exclusive locks on resources.

The **!stacks** extension gives a brief summary of the state of every thread. This can be used instead of the !process extension to get a quick overview of the system, especially when debugging multithread issues, such as resource conflicts or deadlocks.

!processfields

8/26/2019 • 2 minutes to read • Edit Online

The **!processfields** extension displays the names and offsets of the fields within the executive process (EPROCESS) block.

```
!processfields
```

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Unavailable (see the Remarks section)

Additional Information

For information about the EPROCESS block, see *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

This extension command is not available in Windows XP or later versions of Windows. Instead, use the **dt (Display Type)** command to show the EPROCESS structure directly:

```
kd> dt nt!_EPROCESS
```

Here is an example of **!processfields** from a Windows 2000 system:

```
kd> !processfields
 EPROCESS structure offsets:

    Pcb:                            0x0
    ExitStatus:                     0x6c
    LockEvent:                      0x70
    LockCount:                      0x80
    CreateTime:                     0x88
    ExitTime:                       0x90
    LockOwner:                      0x98
    UniqueProcessId:                0x9c
    ActiveProcessLinks:             0xa0
    QuotaPeakPoolUsage[0]:          0xa8
    QuotaPoolUsage[0]:              0xb0
    PagefileUsage:                  0xb8
    CommitCharge:                   0xbc
    PeakPagefileUsage:              0xc0
    PeakVirtualSize:                0xc4
    VirtualSize:                    0xc8
    Vm:                             0xd0
    DebugPort:                      0x120
    ExceptionPort:                  0x124
    ObjectTable:                    0x128
    Token:                          0x12c
    WorkingSetLock:                 0x130
    WorkingSetPage:                 0x150
    ProcessOutswapEnabled:          0x154
    ProcessOutswapped:              0x155
    AddressSpaceInitialized:        0x156
    AddressSpaceDeleted:            0x157
    AddressCreationLock:            0x158
    ForkInProgress:                 0x17c
    VmOperation:                    0x180
    VmOperationEvent:               0x184
    PageDirectoryPte:               0x1f0
    LastFaultCount:                 0x18c
    VadRoot:                        0x194
    VadHint:                        0x198
    CloneRoot:                      0x19c
    NumberOfPrivatePages:           0x1a0
    NumberOfLockedPages:            0x1a4
    ForkWasSuccessful:              0x182
    ExitProcessCalled:              0x1aa
    CreateProcessReported:          0x1ab
    SectionHandle:                  0x1ac
    Peb:                            0x1b0
    SectionBaseAddress:             0x1b4
    QuotaBlock:                     0x1b8
    LastThreadExitStatus:           0x1bc
    WorkingSetWatch:                0x1c0
    InheritedFromUniqueProcessId:   0x1c8
    GrantedAccess:                  0x1cc
    DefaultHardErrorProcessing      0x1d0
    LdtInformation:                 0x1d4
    VadFreeHint:                    0x1d8
    VdmObjects:                     0x1dc
    DeviceMap:                      0x1e0
    ImageFileName[0]:               0x1fc
    VmTrimFaultValue:               0x20c
    Win32Process:                   0x214
    Win32WindowStation:             0x1c4
```

!processirps

12/5/2018 • 2 minutes to read • Edit Online

The **!processirps** extension displays information about I/O request packets (IRPs) associated with processes.

```
!processirps
!processirps ProcessAddress [Flags]
```

Parameters

**** *ProcessAddress*
The address of a process. If you specify *ProcessAddress*, only IRPs associated with that process are displayed. If you do not specify *ProcessAddress*, IRPs for all processes are displayed.

**** *Flags*
A bitwise OR of one or more of the following flags.

Bit 0 (0x1)
Display IRPs queued to threads.

Bit 1 (0x2)
Display IRPs queued to file objects.

If you specify *Flags*, you must also specify *ProcessAddress*. If you do not specify *Flags*, IRPs queued to both threads and file objects are displayed.

DLL

kdexts.dll

Remarks

This command enables you to quickly identify any queued IRPs for a process, both those that are queued to threads and those that are queued to file objects. IRPs are queued to a file object when the file object has a completion port associated with it.

Examples

You can use **!process** command to get process addresses. For example, you could get the process address for explorer.exe.

```
2: kd> !process 0 0
**** NT ACTIVE PROCESS DUMP ****
...
PROCESS fffffa800688c940
    SessionId: 1  Cid: 0bbc    Peb: 7f70da5e000  ParentCid: 0b84
    DirBase: 2db10000  ObjectTable: fffff8a0025bd440  HandleCount: 1056.
    Image: explorer.exe
```

Now you can pass the process address for explorer.exe to the **!processirps** command. The following output shows that explorer.exe has IRPs queued to threads and IRPs queued to file objects.

```
2: kd> !processirps fffffa800688c940
**** PROCESS fffffa800688c940 (Image: explorer.exe) ****

Checking threads for IRPs.

  Thread fffffa800689f080:

    IRP fffffa80045ccc10 - Owned by \FileSystem\Ntfs for device fffffa8004f5c030
    IRP fffffa800454f650 - Owned by \FileSystem\Ntfs for device fffffa8004f5c030
    ...
    IRP fffffa80068e9c10 - Owned by \FileSystem\Ntfs for device fffffa8004f5c030

Checking file objects for IRPs.

  FileObject fffffa80068795e0 (handle 8bc):

    IRP fffffa8006590cf0 - Owned by \Driver\DeviceApi for device DeviceApi (fffffa800363ae40)

  ...

  FileObject fffffa8005bf59c0 (handle 900):

    IRP fffffa8006659010 - Owned by \Driver\DeviceApi for device DeviceApi (fffffa800363ae40)
```

!psp

12/5/2018 • 2 minutes to read • Edit Online

The **!psp** extension displays the processor state parameter (PSP) register at the specified address.

This extension is supported only on Itanium-based target computers.

```
!psp Address [DisplayLevel]
```

Important This command has been deprecated in the Windows Debugger Version 10.0.14257 and later, and is no longer available.

Parameters

Address
Specifies the hexadecimal address of the PSP register to display.

DisplayLevel
Can be any one of the following options:

0
Displays only the values of each PSP field. This is the default.

1
Displays more in-depth information on each of the PSP fields that is not reserved or ignored.

2
Displays more in-depth information on all of the PSP fields, including those that are ignored or reserved.

DLL

Windows 2000	Unavailable
Windows XP and later	Kdexts.dll

!pte

4/8/2019 • 3 minutes to read • Edit Online

The **!pte** extension displays the page table entry (PTE) and page directory entry (PDE) for the specified address.

Syntax

```
!pte VirtualAddress
!pte PTE
!pte LiteralAddress 1
```

Parameters

VirtualAddress
Specifies the virtual address whose page table is desired.

PTE
Specifies the address of an actual PTE.

LiteralAddress **** **1**
Specifies the address of an actual PTE or PDE.

DLL

Kdexts.dll

Additional Information

For information about page tables, page directories, and an explanation of the status bits, see *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

If one parameter is supplied and this parameter is an address in the region of memory where the page tables are stored, the debugger treats this as the *PTE* parameter. This parameter is taken as the actual address of the desired PTE, and the debugger will display this PTE and the corresponding PDE.

If one parameter is supplied and this parameter is not an address in this region, the debugger treats this as the *VirtualAddress* parameter. The PTE and PDE that hold the mapping for this address are displayed.

If two parameters are supplied and the second parameter is **1** (or any other small number), the debugger treats the first parameter as *LiteralAddress*. This address is interpreted as the actual address of a PTE and also as the actual address of a PDE, and the corresponding (and possibly invalid) data will be shown.

(x86 or x64 target computer only) If two parameters are supplied and the second parameter is greater than the first, the debugger treats the two parameters as *StartAddress* and *EndAddress*. The command then displays the PTEs for each page in the specified memory range.

For a list of all system PTEs, use the **!sysptes** extension.

Here is an example from an x86 target computer:

```
kd> !pte 801544f4
801544F4  - PDE at C0300800        PTE at C0200550
          contains 0003B163      contains 00154121
        pfn 3b G-DA--KWV    pfn 154 G--A--KRV
```

The first line of this example restates the virtual address being investigated. It then gives the virtual address of the PDE and the PTE, which contain information about the virtual-physical mapping of this address.

The second line gives the actual contents of the PDE and the PTE.

The third line takes these contents and analyzes them, breaking them into the page frame number (PFN) and the status bits.

See the **!pfn** extension or the Converting Virtual Addresses to Physical Addresses section for information about how to interpret and use the PFN.

On an x86 or x64 target computer, the status bits for the PDE and the PTE are shown in the following table. The **!pte** display indicates these bits with capital letters or dashes, and adds additional information as well.

BIT	DISPLAY WHEN SET	DISPLAY WHEN CLEAR	MEANING
0x200	C	-	Copy on write.
0x100	G	-	Global.
0x80	L	-	Large page. This only occurs in PDEs, never in PTEs.
0x40	D	-	Dirty.
0x20	A	-	Accessed.
0x10	N	-	Cache disabled.
0x8	T	-	Write-through.
0x4	U	K	Owner (user mode or kernel mode).
0x2	W	R	Writeable or read-only. Only on multiprocessor computers and any computer running Windows Vista or later.
0x1	V		Valid.

BIT	DISPLAY WHEN SET	DISPLAY WHEN CLEAR	MEANING
	E	-	Executable page. For platforms that do not support a hardware execute/noexecute bit, including many x86 systems, the E is always displayed.

On an Itanium target computer, the status bits for the PDE and the PTE are slightly different from those of the PPE. The Itanium PPE bits are as follows:

DISPLAY WHEN SET	DISPLAY WHEN CLEAR	MEANING
V		Valid.
U	K	Owner (user mode or kernel mode).
D	-	Dirty.
A	-	Accessed.
W	R	Writeable or read-only. Only on multiprocessor computers and any computer running Windows Vista or later.
E	-	Execute.
C	-	Copy on write.

!pte2va

4/8/2019 • 2 minutes to read • Edit Online

The **!pte2va** extension displays the virtual address that corresponds to the specified page table entry (PTE).

```
!pte2va Address
```

Parameters

Address
Specifies the PTE.

DLL

Windows 2000	Unavailable
Windows XP and later	Kdexts.dll

Additional Information

For information about page tables and PTEs, see *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

To examine the contents of a specific PTE, use the **!pte** extension.

Here is an example of the output from the **!pte2va** extension:

```
kd> !pte2va 9230
000800000248c000
```

!ptov

8/26/2019 • 2 minutes to read • Edit Online

The **!ptov** extension displays the entire physical-to-virtual map for a given process.

```
!ptov DirBase
```

Parameters

DirBase
Specifies the directory base for the process. To determine the directory base, use the **!process** command, and look at the value displayed for DirBase.

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Kdexts.dll

Remarks

Here is an example. First, use **.process** and **!process** to determine the directory base of the current process:

```
1: kd> .process
Implicit process is now 852b4040
1: kd> !process 852b4040 1
PROCESS 852b4040  SessionId: none  Cid: 0004    Peb: 00000000  ParentCid: 0000
    DirBase: 00185000  ObjectTable: 83203000  HandleCount: 663.
    Image: System
    ...
```

In this case, the directory base is 0x00185000. Pass this address to **!ptov**:

```
1: kd> !ptov 185000
X86PtoV: pagedir 185000, PAE enabled.
15e11000 10000
549e6000 20000
...
60a000 210000
40b000 211000
...
54ad3000 25f000
548d3000 260000
...
d71000 77510000
...
```

The numbers in the left column are the physical addresses of each memory page that has a mapping for this process. The numbers in the right column are the virtual addresses to which they are mapped.

The total display is very long.

Here is a 64-bit example.

```
3: kd> .process
Implicit process is now fffffa80`0361eb30
3: kd> !process fffffa80`0361eb30 1
PROCESS fffffa800361eb30
    SessionId: none  Cid: 0004    Peb: 00000000  ParentCid: 0000
    DirBase: 00187000  ObjectTable: fffff8a000002870  HandleCount: 919.
    Image: System
    ...
3: kd> !ptov 187000
Amd64PtoV: pagedir 187000
00000001`034fb000 1d0000
a757c000 1d1000
00000001`0103d000 1d2000
c041e000 1d3000
...
2ed6f000 fffff680`00001000
00000001`13939000 fffff680`00003000
ceefb000 fffff680`00008000
...
```

The directory base is the physical address of the first table that is used in virtual address translation. This table has different names depending on the bitness of the target operating system and whether Physical Address Extension (PAE) is enabled for the target operating system.

For 64-bit Windows, the directory base is the physical address of the Page Map Level 4 (PML4) table. For 32-bit Windows with PAE enabled, the directory base is the physical address of the Page Directory Pointers (PDP) table. For 32-bit Windows with PAE disabled, the directory bas is the physical address of the Page Directory (PD) table.

For related topics, see **!vtop** and Converting Virtual Addresses to Physical Addresses. For information about virtual address translation, see *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

!qlocks

12/5/2018 • 2 minutes to read • Edit Online

The **!qlocks** extension displays the state of all queued spin locks.

```
!qlocks
```

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Kdexts.dll

Additional Information

For information about spin locks, see the Windows Driver Kit (WDK) documentation and *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

This command is useful only on a multiprocessor system.

Here is an example:

```
0: kd> !qlocks
Key: O = Owner, 1-n = Wait order, blank = not owned/waiting, C = Corrupt

                      Processor Number
    Lock Name         0  1  2  3

KE   - Dispatcher
KE   - Unused Spare
MM   - PFN
MM   - System Space
CC   - Vacb
CC   - Master
EX   - NonPagedPool
IO   - Cancel
EX   - WorkQueue
IO   - Vpb
IO   - Database
IO   - Completion
NTFS - Struct
AFD  - WorkQueue
CC   - Bcb
MM   - MM NonPagedPool
```

!ready

4/8/2019 • 2 minutes to read • Edit Online

The **!ready** extension displays summary information about each thread in the system in a READY state.

```
!ready [Flags]
```

Parameters

Flags
Specifies the level of detail to display. *Flags* can be any combination of the following bits. If *Flags* is 0, only a minimal amount of information is displayed. The default is 0x6.

Bit 1 (0x2)
Causes the display to include the thread's wait states.

Bit 2 (0x4)
If this is included without Bit 1 (0x2), this has no effect. If this is included along with Bit 1, the thread is displayed with a stack trace.

Bit 3 (0x8)
Causes the display of each function to include the return address, the stack pointer, and (on Itanium systems) the **bsp** register value. The display of function arguments is suppressed.

Bit 4 (0x10)
Causes the display of each function to include only the return address; arguments and stack pointers are suppressed.

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Kdexts.dll

Additional Information

For information about thread scheduling and the READY state, see *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

The output from this extension is similar to that of **!thread**, except that only ready threads are displayed, and they are sorted in order of decreasing priority.

!reg

6/26/2019 • 7 minutes to read • Edit Online

The **!reg** extension displays and searches through registry data.

```
!reg {querykey|q} FullKeyPath
!reg keyinfo HiveAddress KeyNodeAddress
!reg kcb Address
!reg knode Address
!reg kbody Address
!reg kvalue Address
!reg valuelist HiveAddress KeyNodeAddress
!reg subkeylist HiveAddress KeyNodeAddress
!reg baseblock HiveAddress
!reg seccache HiveAddress
!reg hashindex [HiveAddress]HashKey
!reg openkeys {HiveAddress|0}
!reg openhandles {HiveAddress|0}
!reg findkcb FullKeyPath
!reg hivelist
!reg viewlist HiveAddress
!reg freebins HiveAddress
!reg freecells BinAddress
!reg dirtyvector HiveAddress
!reg cellindex HiveAddress Index
!reg freehints HiveAddress Storage Display
!reg translist {RmAddress|0}
!reg uowlist TransactionAddress
!reg locktable KcbAddress ThreadAddress
!reg convkey KeyPath
!reg postblocklist
!reg notifylist
!reg ixlock LockAddress
!reg dumppool [s|r]
```

Parameters

{**querykey**|**q**} **** *FullKeyPath*
Displays subkeys and values of a key if the key is cached. *FullKeyPath* specifies the full key path.

keyinfo *HiveAddress* **** *KeyNodeAddress*
Displays subkeys and values of a key node. *HiveAddress* specifies the address of the hive. *KeyNodeAddress* specifies the address of the key node.

kcb **** *Address*
Displays a registry key control block. *Address* specifies the address of the key control block.

knode **** *Address*
Displays a registry key node structure. *Address* specifies the address of the key node.

kbody **** *Address*
Displays a registry key body structure. *Address* specifies the address of the key body. (Registry key bodies are the actual objects associated with handles.)

kvalue **** *Address*
Displays a registry key value structure. *Address* specifies the address of the value.

valuelist **** *HiveAddress* **** *KeyNodeAddress*
Displays a list of the values in the specified key node. *HiveAddress* specifies the address of the hive. *KeyNodeAddress* specifies the address of the key node.

subkeylist **** *HiveAddress* **** *KeyNodeAddress*
Displays a list of the subkeys of the specified key node. *HiveAddress* specifies the address of the hive. *KeyNodeAddress* specifies the address of the key node.

baseblock **** *HiveAddress*
Displays the base block for a hive (also known as the *hive header*). *HiveAddress* specifies the address of the hive.

seccache **** *HiveAddress*
Displays the security cache for a hive. *HiveAddress* specifies the address of the hive.

hashindex **** [*HiveAddress*] **** *HashKey*
Computes the hash index entry for a hash key. *HiveAddress* specifies the address of the hive. *HashKey* specifies the key.

Note *HiveAddress* is required if the target computer is running Windows 7 or later.

openkeys {*HiveAddress*|**0**}
Displays all open keys in a hive. *HiveAddress* specifies the address of the hive. If zero is used instead, the entire registry hash table is displayed; this table contains all open keys in the registry.

findkcb **** *FullKeyPath*
Displays the registry key control block corresponding to a registry path. *FullKeyPath* specifies the full key path; this path must be present in the hash table.

hivelist
Displays a list of all hives in the system, along with detailed information about each hive.

viewlist **** *HiveAddress*
Displays all pinned and mapped views for a hive, with detailed information for each view. *HiveAddress* specifies the address of the hive.

freebins **** *HiveAddress*
Displays all free bins for a hive, with detailed information for each bin. *HiveAddress* specifies the address of the hive.

freecells **** *BinAddress*
Iterates through a bin and displays all free cells inside it. *BinAddress* specifies the address of the bin.

dirtyvector **** *HiveAddress*
Displays the dirty vector for a hive. *HiveAddress* specifies the address of the hive.

cellindex **** *HiveAddress* **** *Index*
Displays the virtual address for a cell in a hive. *HiveAddress* specifies the address of the hive. *Index* specifies the cell index.

freehints *HiveAddress* **** *Storage* **** *Display*
Displays free hint information.

translist {*RmAddress*|**0**}
Displays the list of active transactions in an RM. *RmAddress* specifies the address of the RM.

uowlist *TransactionAddress*
Displays the list of UoWs attached to a transaction. *TransactionAddress* specifies the address of the transaction.

locktable *KcbAddress ThreadAddress*

Displays relevant lock table content.

convkey *KeyPath*
Displays hash keys for a key path.

postblocklist
Displays the list of threads that have postblocks posted.

notifylist
Displays the list of notify blocks in the system.

ixlock *LockAddress*
Displays ownership of an intent lock. *LockAddress* specifies the address of the lock.

dumppool [**s**|**r**]
Displays registry-allocated paged pool. If **s** is specified, the list of registry pages is saved to a temporary file. If **r** is specified, the registry page list is restored from the previously saved temporary file.

DLL

Kdexts.dll

Additional Information

For information about the registry and its components, see *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

Here is an example. First use **!reg hivelist** to get a list of hive addresses.

```
00: kd> !reg hivelist
##

## |     HiveAddr     |Stable Length|    Stable Map    |Volatile Length|    Volatile Map
|MappedViews|PinnedViews|U(Cnt)|     BaseBlock     | FileName

| fffff8a000014010 |       1000  | fffff8a0000140b0 |       1000    |  fffff8a000014328  |     0|
fffff8a00001e000  | <NONAME>
| fffff8a000028010 |     a15000  | fffff8a00002e000 |      1a000    |  fffff8a000028328  |     0|
fffff8a000029000  | SYSTEM
| fffff8a00004f010 |      14000  | fffff8a00004f0b0 |       c000    |  fffff8a00004f328  |     0|
fffff8a000050000  | <NONAME>
| fffff8a000329010 |       6000  | fffff8a0003290b0 |          0    |  0000000000000000  |     0|
fffff8a00032f000  | Device\HarddiskVolume1\Boot\BCD
| fffff8a0002f2010 |    4255000  | fffff8a0006fa000 |       6000    |  fffff8a0002f2328  |     0|
fffff8a00036c000  | emRoot\System32\Config\SOFTWARE
| fffff8a000df0010 |      f7000  | fffff8a000df00b0 |       1000    |  fffff8a000df0328  |     0|
fffff8a000df1000  | temRoot\System32\Config\DEFAULT
| fffff8a0010f8010 |       9000  | fffff8a0010f80b0 |       1000    |  fffff8a0010f8328  |     0|
fffff8a0010f9000  | emRoot\System32\Config\SECURITY
| fffff8a001158010 |       7000  | fffff8a0011580b0 |          0    |  0000000000000000  |     0|
fffff8a001159000  | \SystemRoot\System32\Config\SAM
| fffff8a00124b010 |      24000  | fffff8a00124b0b0 |          0    |  0000000000000000  |     0|
fffff8a00124c000  | files\NetworkService\NTUSER.DAT
| fffff8a0012df220 |      b7000  | fffff8a0012df2c0 |          0    |  0000000000000000  |     0|
fffff8a0012e6000  | \SystemRoot\System32\Config\BBI
| fffff8a001312220 |      26000  | fffff8a0013122c0 |          0    |  0000000000000000  |     0|
fffff8a00117e000  | rofiles\LocalService\NTUSER.DAT
| fffff8a001928010 |      64000  | fffff8a0019280b0 |       3000    |  fffff8a001928328  |     0|
fffff8a00192b000  | User.MYTESTCOMPUTER2\ntuser.dat
| fffff8a001b9b010 |     203000  | fffff8a001bc4000 |          0    |  0000000000000000  |     0|
fffff8a001b9c000  | \Microsoft\Windows\UsrClass.dat
| fffff8a001dc0010 |      30000  | fffff8a001dc00b0 |          0    |  0000000000000000  |     0|
fffff8a001dc2000  | Volume Information\Syscache.hve
## | fffff8a0022dc010 |     175000  | fffff8a0022dc0b0 |          0    |  0000000000000000  |     0|
fffff8a0022dd000  | \AppCompat\Programs\Amcache.hve
```

Use the third hive address in the preceding output (fffff8a00004f010) as an argument to **!reg openkeys**.

```
0: kd> !reg openkeys fffff8a00004f010

# Hive: \REGISTRY\MACHINE\HARDWARE

Index e9:    3069276d kcb=fffff8a00007eb98 cell=00000220 f=00200000
\REGISTRY\MACHINE\HARDWARE\DESCRIPTION\SYSTEM
Index 101:   292eea1f kcb=fffff8a00007ecc0 cell=000003b8 f=00200000
\REGISTRY\MACHINE\HARDWARE\DESCRIPTION\SYSTEM\MULTIFUNCTIONADAPTER
Index 140:   d927b0d4 kcb=fffff8a00007ea70 cell=000001a8 f=00200000 \REGISTRY\MACHINE\HARDWARE\DESCRIPTION
Index 160:   96d26a30 kcb=fffff8a00007e6f8 cell=00000020 f=002c0000 \REGISTRY\MACHINE\HARDWARE

# 0x4 keys found
```

Use the first full key path in the preceding output (\REGISTRY\MACHINE\HARDWARE\DESCRIPTION\SYSTEM) as an argument to **!reg querykey**.

```
0: kd> !reg querykey \REGISTRY\MACHINE\HARDWARE\DESCRIPTION\SYSTEM

Found KCB = fffff8a00007eb98 :: \REGISTRY\MACHINE\HARDWARE\DESCRIPTION\SYSTEM

Hive         fffff8a00004f010
KeyNode      fffff8a000054224

[SubKeyAddr]         [SubKeyName]
fffff8a000060244     CentralProcessor
fffff8a00006042c     FloatingPointProcessor
fffff8a0000543bc     MultifunctionAdapter

[SubKeyAddr]         [VolatileSubKeyName]
fffff8a000338d8c     BIOS
fffff8a0002a2e4c     VideoAdapterBusses

 Use '!reg keyinfo fffff8a00004f010 <SubKeyAddr>' to dump the subkey details

[ValueType]         [ValueName]                   [ValueData]
REG_BINARY          Component Information         0x542AC - 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
REG_SZ              Identifier                    AT/AT COMPATIBLE
REG_FULL_RESOURCE_DESCRIPTORConfiguration Data           ff ff ff ff ff ff ff ff 00 00 00 00 02 00 00 00 05
00 00 00 18 00 00 00 00 00 00 00 00 00 00 00 00 00 00 80 00 ff 03 00 00 3f 00 fe 00 02 00 81 00 fe 03 00 00
3f 00 fe 00 02 00 05 00 00 00 08 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0c 00 00 00 04 00
REG_SZ              SystemBiosDate                07/18/07
REG_MULTI_SZ        SystemBiosVersion             HPQOEM - 20070718\0\0
REG_SZ              VideoBiosDate                 03/23/20
REG_MULTI_SZ        VideoBiosVersion              Hardware Version 0.0\0\0
```

Here is another example:

```
kd> !reg hivelist
##

## | HiveAddr |Stable Length|Stable Map|Volatile Length|Volatile Map|MappedViews|PinnedViews|U(Cnt)| BaseBlock
| FileName

| e16e7428 |       2000  | e16e7484 |          0    |  00000000  |        1  |        0  |     0| e101f000  |
\Microsoft\Windows\UsrClass.dat
| e1705a78 |      77000  | e1705ad4 |       1000    |  e1705bb0  |       30  |        0  |     0| e101c000  |
ttings\Administrator\ntuser.dat
| e13d4b88 |     814000  | e146a000 |       1000    |  e13d4cc0  |      255  |        0  |     0| e1460000  |
emRoot\System32\Config\SOFTWARE
| e13ad008 |      23000  | e13ad064 |       1000    |  e13ad140  |        9  |        0  |     0| e145e000  |
temRoot\System32\Config\DEFAULT
| e13b3b88 |       a000  | e13b3be4 |       1000    |  e13b3cc0  |        3  |        0  |     0| e145d000  |
emRoot\System32\Config\SECURITY
| e142d008 |       5000  | e142d064 |          0    |  00000000  |        2  |        0  |     0| e145f000  |
<UNKNOWN>
| e11e3628 |       4000  | e11e3684 |       3000    |  e11e3760  |        0  |        0  |     0| e11e4000  |
<NONAME>
| e10168a8 |     1c1000  | e1016904 |      15000    |  e10169e0  |       66  |        0  |     0| e1017000  |
SYSTEM
## | e10072c8 |       1000  | e1007324 |          0    |  00000000  |        0  |        0  |     0| e1010000
| <NONAME>


kd> !reg hashindex e16e7428

CmpCacheTable = e100a000

Hash Index[e16e7428] : 5ac
Hash Entry[e16e7428] : e100b6b0

kd> !reg openkeys e16e7428

Index 68:  7bab7683 kcb=e13314f8 cell=00000740 f=00200004 \REGISTRY\USER\S-1-5-21-1715567821-413027322-
527237240-500_Classes\CLSID
Index 7a1:  48a30288 kcb=e13a3738 cell=00000020 f=002c0004 \REGISTRY\USER\S-1-5-21-1715567821-413027322-
527237240-500_Classes
```

To display formatted registry key information, use the **!dreg** extension instead.

!regkcb

12/5/2018 • 2 minutes to read • Edit Online

The **!regkcb** extension displays a registry key control block.

```
!regkcb Address
```

Parameters

Address
Specifies the address of the key control block.

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Unavailable

Additional Information

For information about the registry and its components, see *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

In Windows 2000, **!regkcb** displays a specific registry key control block.

In Windows XP and later versions of Windows, the **!reg** extension command should be used instead.

Every registry key has a control block that contains properties, such as its permissions.

!rellist

12/5/2018 • 2 minutes to read • Edit Online

The **!rellist** extension displays a Plug and Play relation list.

```
!rellist Address [Flags]
```

Parameters

Address
Specifies the address of the relation list.

Flags
Specifies additional information to include in the display. This can be any combination of the following bits (the default is zero):

Bit 1 (0x2)
Causes the display to include the CM_RESOURCE_LIST. The display also includes the boot resources list, if it is available.

Bit 2 (0x4)
Causes the display to include the resource requirements list (IO_RESOURCE_LIST).

Bit 3 (0x8)
Causes the display to include the translated CM_RESOURCE_LIST.

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Kdexts.dll

Additional Information

See Plug and Play Debugging for applications of this extension command. For information about these list structures, see the Windows Driver Kit (WDK) documentation.

!ruleinfo

12/5/2018 • 2 minutes to read • Edit Online

The **!ruleinfo** command displays information about a Driver Verifier rule.

```
!ruleinfo RuleId [RuleState [SubState]]
```

Parameters

RuleId
The ID of the verifier rule. This is the first argument of the **DRIVER_VERIFIER_DETECTED_VIOLATION** bug check.

RuleState
Additional state information about the violation. This is the third argument of the **DRIVER_VERIFIER_DETECTED_VIOLATION** bug check.

SubState
Sub-state information about the violation. This is the fourth argument of the **DRIVER_VERIFIER_DETECTED_VIOLATION** bug check.

DLL

ext.dll

Remarks

This command applies only to rules in the Driver Verifier extension; that is, rules that have an ID greater than or equal to 0x10000.

The following example shows the four arguments of a **DRIVER_VERIFIER_DETECTED_VIOLATION** bug check.

```
DRIVER_VERIFIER_DETECTED_VIOLATION (c4)
...
Arguments:
Arg1: 0000000000091001, ID of the 'NdisOidComplete' rule that was violated.
Arg2: fffff800002d49d0, A pointer to the string describing the violated rule condition.
Arg3: ffffe000027b8370, Address of internal rule state (second argument to !ruleinfo).
Arg4: ffffe000027b83f8, Address of supplemental states (third argument to !ruleinfo).

## Debugging Details:


DV_VIOLATED_CONDITION:  This OID should only be completed with NDIS_STATUS_NOT_ACCEPTED,
                        NDIS_STATUS_SUCCESS, or NDIS_STATUS_PENDING.

DV_MSDN_LINK: https://go.microsoft.com/fwlink/p/?linkid=278802

DRIVER_OBJECT: ffffe0000277a2b0
...

STACK_TEXT:
ffffd000`2118ff58 fffff803`4c83afa2 : 00000000`000000c4 00000000`00000001 ...
ffffd000`2118ff60 fffff803`4c83a8c0 : 00000000`00000003 00000000`00091001 ...
...

STACK_COMMAND:  kb

FOLLOWUP_NAME:  Xxxx

FAILURE_BUCKET_ID:  Xxxx
...
```

In the preceding output, the rule ID (0x91001) is shown as Arg1. Arg3 and Arg4 are the addresses of rule state and substate information. You can pass the rule ID, the rule state, and the substate to **!ruleinfo** to get a description of the rule and a link to detailed documentation of the rule.

```
3: kd> !ruleinfo 0x91001 0xffffe000027b8370 0xffffe000027b83f8

RULE_ID: 0x91001

RULE_NAME: NdisOidComplete

RULE_DESCRIPTION:
This rule verifies if an NDIS miniport driver completes an OID correctly.
Check RULE_STATE for Oid ( use !ndiskd.oid ), which can be one of the following:
1) NULL,
2) Pending OID, or
3) Previous OID if no OID is pending.

MSDN_LINK: https://go.microsoft.com/fwlink/p/?linkid=278802

CONTEXT: Miniport 0xFFFFE0000283F1A0

CURRENT_TIME (Timed Rules): 142 seconds

RULE_STATE: 0xFFFFE000027B83F8
```

!running

12/5/2018 • 2 minutes to read • Edit Online

The **!running** extension displays a list of running threads on all processors of the target computer.

```
!running [-i] [-t]
```

Parameters

-i

Causes the display to include idle processors as well.

-t

Causes a stack trace to be displayed for each processor.

DLL

Windows 2000	Unavailable
Windows XP and later	Kdexts.dll

Additional Information

For more information about debugging multiprocessor computers, see Multiprocessor Syntax.

Remarks

With no options, **!running** will display the affinity of all active processors and all idle processors. For all active processors, it will also display the current and next thread fields from the process control block (PRCB) and the state of the 16 built-in queued spin locks.

Here is an example from a multiprocessor Itanium system:

```
0: kd> !running

System Processors 3 (affinity mask)
 Idle Processors 0

     Prcb              Current           Next
  0  e0000000818f8000  e0000000818f9e50  e0000000866f12f0  ................
  1  e000000086f16010  e00000008620ebe0  e000000086eddbc0  .O..............
```

The 16 characters at the end of each line indicate the built-in queued spin locks (the LockQueue entries in the PRCB). A period (.) indicates that the lock is not in use, **O** means the lock is owned by this processor, and **W** means the processor is queued for the lock. To see more information about the spin lock queue, use **!qlocks**.

Here is an example that shows active and idle processors, along with their stack traces:

```
0: kd> !running -it

System Processors f (affinity mask)
  Idle Processors f
All processors idle.

     Prcb      Current   Next
  0  ffdff120  805495a0            .............

ChildEBP RetAddr
8053e3f0 805329c2 nt!RtlpBreakWithStatusInstruction
8053e3f0 80533464 nt!_KeUpdateSystemTime+0x126
ffdff980 ffdff980 nt!KiIdleLoop+0x14

 1  f87e0120  f87e2e60            .............

ChildEBP RetAddr
f87e0980 f87e0980 nt!KiIdleLoop+0x14

 2  f87f0120  f87f2e60            .............

ChildEBP RetAddr
f87f0980 f87f0980 nt!KiIdleLoop+0x14

  3  f8800120  f8802e60            .............

ChildEBP RetAddr
f8800980 f8800980 nt!KiIdleLoop+0x14
```

!scm

8/26/2019 • 2 minutes to read • Edit Online

The **!scm** extension displays the specified shared cache map.

```
!scm Address
```

Parameters

Address
Specifies the address of the shared cache map.

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Unavailable

Additional Information

For information about cache management, see the Microsoft Windows SDK documentation and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

For information about other cache management extensions, see the **!cchelp** extension.

Remarks

In Windows XP and later versions of Windows, use the **dt nt!_SHARED_CACHE_MAP Address** command instead of **!scm**.

!search

12/5/2018 • 3 minutes to read • Edit Online

The **!search** extension searches pages in physical memory for pointer-sized data that matches the specified criteria.

Syntax

```
!search [-s] [-p] Data [ Delta [ StartPFN [ EndPFN ]]]
!search -?
```

Parameters

-s
Causes symbol check errors to be ignored during the search. This is useful if you are getting too many "incorrect symbols for kernel" errors.

-p
Causes the value of *Data* to be interpreted as a 32-bit value, preventing any sign extension.

Data
Specifies the data to search for. *Data* must be the size of a pointer on the target system (32 bits or 64 bits). An exact match for the value of *Data* is always displayed. Other matches are displayed as well, depending on the value of *Delta*; see the Remarks section below for details.

Delta
Specifies the allowable difference between a value in memory and the value of *Data*. See the Remarks section below for details.

StartPFN
Specifies the page frame number (PFN) of the beginning of the range to be searched. If this is omitted, the search begins at the lowest physical page.

EndPFN
Specifies the page frame number (PFN) of the end of the range to be searched. If this is omitted, the search ends at the highest physical page.

-?
Displays help for this extension in the Debugger Command window.

DLL

Kdexts.dll

Additional Information

For more ways to display and search physical memory, see Reading and Writing Memory.

Remarks

If *StartPFN* and *EndPFN* are specified, these are taken as the page frame numbers of the beginning and end of the range in physical memory to be searched. For an explanation of page frame numbers, see Converting Virtual Addresses to Physical Addresses. If *StartPFN* and *EndPFN* are omitted, all physical memory is searched.

All hits are displayed.

The **!search** extension will search through all memory for in the specified page range and examine each ULONG_PTR-aligned value. Values that satisfy at least one of the following criteria are displayed:

- The value matches *Data* exactly.
- If Delta is 0 or omitted: The value differs from *Data* by a single bit.
- If Delta is nonzero: The value differs from *Data* by at most *Delta*. In other words, the value lies in the range [Data - Delta, Data + Delta].
- If Delta is nonzero: The value differs from the lowest number in the range (Data - Delta) by a single bit.

In most cases, *Data* will specify an address you are interested in, but any ULONG_PTR sized data can be specified.

Because the debugger's search engine structures reside in memory on the target computer, if you search all of memory (or any range containing these structures) you will see matches in the area where the structures themselves are located. If you need to eliminate these matches, do a search for a random value; this will indicate where the debugger's search structures are located.

Here are some examples. The following will search the memory page with PFN 0x237D for values between 0x80001230 and 0x80001238, inclusive:

```
kd> !search 80001234 4 237d 237d
```

The following will search the memory pages ranging from PFN 0x2370 to 0x237F for values that are within one bit of 0x0F100F0F. The exact matches are indicated in bold; the others are off by one bit:

```
kd> !search 0f100f0f 0 2370 237f
Searching PFNs in range 00002370 - 0000237F for [0F100F0F - 0F100F0F]

Pfn      Offset   Hit      Va       Pte
- - - - - - - - - - - - - - - - - -
0000237B 00000368 0F000F0F 01003368 C0004014
0000237C 00000100 0F100F0F 01004100 C0004014
0000237D 000003A8 0F100F0F 010053A8 C0004014
0000237D 000003C8 0F100F8F 010053C8 C0004014
0000237D 000003E8 0F100F0F 010053E8 C0004014
0000237D 00000408 0F100F0F 01005408 C0004014
0000237D 00000428 0F100F8F 01005428 C0004014
Search done.
```

The columns in the display are as follows: **Pfn** is the page frame number (PFN) of the page; **Offset** is the offset on that page; **Hit** is the value at that address; **Va** is the virtual address mapped to this physical address (if this exists and can be determined); **Pte** is the page table entry (PTE).

To calculate the physical address, shift the PFN left three hexadecimal digits (12 bits) and add the offset. For example, the last line in the table is virtual address 0x0237D000 + 0x428 = 0x02347D428.

!searchpte

4/8/2019 • 2 minutes to read • Edit Online

The **!searchpte** extension searches physical memory for the specified page frame number (PFN).

```
!searchpte PFN
!searchpte -?
```

Parameters

PFN
Specifies the PFN in hexadecimal format.

-?
Displays help for this extension in the Debugger Command window.

DLL

Windows 2000	Unavailable
Windows XP and later	Kdexts.dll

Additional Information

For information about page tables and page directories, see *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

To stop execution at any time, press CTRL+BREAK (in WinDbg) or CTRL+C (in KD).

!sel

12/5/2018 • 2 minutes to read • Edit Online

The **!sel** extension command is obsolete. Use the **dg (Display Selector)** command instead.

!session

12/5/2018 • 2 minutes to read • Edit Online

The **!session** extension controls the session context. It can also display a list of all user sessions.

Syntax

```
!session
!session -s DefaultSession
!session -?
```

Parameters

-s **** *DefaultSession*
Sets the session context to the specified value. If *DefaultSession* is -1, the session context is set to the current session.

-?
Displays help for this extension in the Debugger Command window.

DLL

Kdexts.dll

Additional Information

For information about user sessions and the Session Manager (smss.exe), see *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

The **!session** extension is used to control the session context. Using **!session** with no parameters will display a list of active sessions on the target computer. Using **!session /s** *DefaultSession* will change the session context to the new default value.

When you set the session context, the process context is automatically changed to the active process for that session, and the **.cache forcedecodeptes** option is enabled so that session addresses are translated properly.

For more details and a list of all the session-related extensions that are affected by the session context, see Changing Contexts.

!smt

12/5/2018 • 2 minutes to read • Edit Online

The **!smt** extension displays a summary of the simultaneous multithreaded processor information.

```
!smt
```

DLL

Windows 2000	Unavailable
Windows XP and later	Kdexts.dll

Remarks

Here is an example:

```
lkd> !smt
SMT Summary:
------------
   KeActiveProcessors: **------------------------------ (00000003)
        KiIdleSummary: -------------------------------- (00000000)
 No PRCB     Set Master SMT Set                                IAID
  0 820f4820 Master     **------------------------------ (00000003)  00
  1 87a4d120 820f4820   **------------------------------ (00000003)  01

Maximum cores per physical processor:   2
Maximum logical processors per core:    1
```

The **No** column indicates the number of the processor.

The **PRCB** column indicates the address of the processor control block for the processor. Each logical processor is listed separately.

Each physical processor has exactly one logical processor listed as the **Master** under the **Set Master** column.

The **SMT Set** column lists the processor's simultaneous multithreaded processor set information.

The **IAID** column lists the initial Advanced Programmable Interrupt Controller identifier (APIC ID). On a true x64 computer, each processor starts with a hard-coded initial APIC ID. This ID value can be retrieved through the CPUID instruction. On certain other computers, the initial APIC ID is not necessarily unique across all processors, so the APIC ID that is accessible through the APIC's memory-mapped input/output (MMIO) space can be modified. This technique enables software to allocate unique APIC IDs for all processors within the computer. Depending on the target computer's processors, the **IAID** column may show this ID or may be blank.

!sprocess

4/8/2019 • 2 minutes to read • Edit Online

The **!sprocess** extension displays information about the specified session process, or about all processes in the specified session.

```
!sprocess Session [Flags [ImageName]]
!sprocess -?
```

Parameters

Session
Specifies the session that owns the desired process. *Session* is always interpreted as a decimal number.

Session can have the following values:

-1	Use current session. This is the default.
-2	Use session context.
-4	Display all processes by session.

Flags
Specifies the level of detail in the display. *Flags* can be any combination of the following bits. The default is 0.

0x0	Display minimal information.
Bit 0 (0x1)	Display time and priority statistics.
Bit 1 (0x2)	Adds to the display a list of threads and events associated with the process and the wait states of the threads.
Bit 2 (0x4)	Adds to the display a list of threads associated with the process. If this bit is used without Bit 1 (0x2), each thread is displayed on a single line. If this is included with Bit 1, each thread is displayed with a stack trace.
Bit 3 (0x8)	Adds to the display of each function the return address, the stack pointer and, on Itanium-based systems, the **bsp** register value. It suppresses the display of function arguments.
Bit 4 (0x10)	Display only the return address of each function. Suppress the arguments and stack pointers.

ImageName
Specifies the name of the process to be displayed. All processes whose executable image names match *ImageName* are displayed. The image name must match that in the EPROCESS block. In general, this is the executable name that was invoked to start the process, including the file extension (usually .exe), and truncated after the fifteenth character. There is no way to specify an image name that contains a space.

-?
Displays help for this extension in the Debugger Command window. This help text has some omissions.

DLL

Windows 2000	Unavailable
Windows XP and later	Kdexts.dll

Additional Information

For information about sessions and processes in kernel mode, see Changing Contexts. For more information about analyzing processes and threads, see *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

The output of this extension is similar to that of **!process**, except that the addresses of _MM_SESSION_SPACE and _MMSESSION are displayed as well.

!srb

12/5/2018 • 2 minutes to read • Edit Online

The **!srb** extension displays information about a SCSI Request Block (SRB).

```
!srb Address
```

Parameters

Address
Specifies the hexadecimal address of the SRB on the target computer.

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Kdexts.dll

Additional Information

For information about SRBs, see the Windows Driver Kit (WDK) documentation.

Remarks

An SRB is a system-defined structure used to communicate I/O requests from a SCSI class driver to a SCSI port driver.

!stacks

4/8/2019 • 2 minutes to read • Edit Online

The **!stacks** extension displays information about the kernel stacks.

Syntax

```
!stacks [Detail [FilterString]]
```

Parameters

Detail
Specifies the level of detail to use in the display. The following table lists the valid values for *Detail*.

0	Displays a summary of the current kernel stacks. This is the default value.
1	Displays stacks that are currently paged out, as well as the current kernel stacks.
2	Displays the full parameters for all stacks, as well as stacks that are currently paged out and the current kernel stacks.

FilterString
Displays only threads that contain the specified substring in a symbol.

DLL

Kdexts.dll

Additional Information

For information about kernel stacks, see *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

The **!stacks** extension gives a brief summary of the state of every thread. You can use this extension instead of the **!process** extension to get a quick overview of the system, especially when debugging multithread issues such as resource conflicts or deadlocks.

The **!findstack** user-mode extension also displays information about particular stacks.

Here is an example of the simplest **!stacks** display:

```
kd> !stacks 0
Proc.Thread  .Thread  ThreadState  Blocker
                                   [System]
   4.000050  827eea10  Blocked    +0xfe0343a5

                                   [smss.exe]

                                   [csrss.exe]
  b0.0000a8  82723b70  Blocked    ntoskrnl!_KiSystemService+0xc4
  b0.0000c8  82719620  Blocked    ntoskrnl!_KiSystemService+0xc4
  b0.0000d0  827d5d50  Blocked    ntoskrnl!_KiSystemService+0xc4
.....
```

The first column shows the process ID and the thread ID (separated by a period).

The second column is the current address of the thread's ETHREAD block.

The third column shows the state of the thread (initialized, ready, running, standby, terminated, transition, or blocked).

The fourth column shows the top address on the thread's stack.

Here are examples of more detailed **!stacks** output:

```
kd> !stacks 1
Proc.Thread  .Thread  ThreadState  Blocker
                                   [System]
   4.000008  827d0030  Blocked    ntoskrnl!MmZeroPageThread+0x66
   4.000010  827d0430  Blocked    ntoskrnl!ExpWorkerThread+0x189
   4.000014  827cf030  Blocked    Stack paged out
   4.000018  827cfda0  Blocked    Stack paged out
   4.00001c  827cfb10  Blocked    ntoskrnl!ExpWorkerThread+0x189
.....
                                   [smss.exe]
  9c.000098  82738310  Blocked    Stack paged out
  9c.0000a0  826a5190  Blocked    Stack paged out
  9c.0000a4  82739d30  Blocked    Stack paged out

                                   [csrss.exe]
  b0.0000bc  826d0030  Blocked    Stack paged out
  b0.0000b4  826c9030  Blocked    Stack paged out
  b0.0000a8  82723b70  Blocked    ntoskrnl!_KiSystemService+0xc4
.....

kd> !stacks 2
Proc.Thread  .Thread  ThreadState  Blocker
                                   [System]
   4.000008  827d0030  Blocked    ntoskrnl!KiSwapThread+0xc5
                                  ntoskrnl!KeWaitForMultipleObjects+0x2b4
                                  ntoskrnl!MmZeroPageThread+0x66
                                  ntoskrnl!Phase1Initialization+0xd82
                                  ntoskrnl!PspSystemThreadStartup+0x4d
                                  ntoskrnl!CreateSystemRootLink+0x3d8
                                  +0x3f3f3f3f
   4.000010  827d0430  Blocked    ntoskrnl!KiSwapThread+0xc5
                                  ntoskrnl!KeRemoveQueue+0x191
.....
```

!swd

12/5/2018 • 2 minutes to read • Edit Online

The **!swd** extension displays the software watchdog timer states for the specified processor, including the deferred procedure call (DPC) and the watchdog timer states for threads.

```
!swd [Processor]
```

Parameters

Processor
Specifies the processor. If *Processor* is omitted, information is displayed for all processors on the target computer.

DLL

Windows 2000	Unavailable
Windows XP and later	Kdexts.dll

Remarks

The watchdog timer shuts down or restarts Windows if Windows stops responding. The times are displayed in seconds.

!sysinfo

12/5/2018 • 2 minutes to read • Edit Online

The **!sysinfo** extension reads and displays specified SMBIOS, Advanced Configuration and Power Interface (ACPI), and CPU information from a dump file or live system.

```
!sysinfo cpuinfo [-csv [-noheaders]]
!sysinfo cpumicrocode [-csv [-noheaders]]
!sysinfo cpuspeed [-csv [-noheaders]]
!sysinfo gbl [-csv [-noheaders]]
!sysinfo machineid [-csv [-noheaders]]
!sysinfo registers
!sysinfo smbios [-csv [-noheaders]] {-debug | -devices | -memory | -power | -processor | -system | -v}
!sysinfo -?
```

Parameters

cpuinfo
Displays information about the processor.

cpumicrocode
(GenuineIntel processors only) Displays the initial and cached microcode processor versions.

cpuspeed
Displays the maximum and current processor speeds.

gbl
Displays the BIOS list of ACPI tables.

machineid
Displays machine ID information for the SMBIOS, BIOS, firmware, system, and baseboard.

registers
Displays machine-specific registers (MSRs).

smbios
Displays the SMBIOS table.

-csv
Displays all data in comma-separated, variable-length (CSV) format.

-noheaders
Suppresses the header for the CSV format.

-debug
Displays output in standard format and CSV format.

-devices
Displays the device entries in the SMBIOS table.

-memory
Displays the memory entries in the SMBIOS table.

-power
Displays the power entries in the SMBIOS table.

-processor
Displays the processor entries in the SMBIOS table.

-system
Displays the system entries in the SMBIOS table.

-v
Verbose. Displays the details of entries in the SMBIOS table.

-?
Displays help for this extension in the Debugger Command window.

DLL

Windows 2000	Unavailable
Windows XP base system **Windows 2003 base system**	Unavailable
Windows XP, Service Pack 2 and later **Windows 2003, Service Pack 1 and later**	Kdexts.dll

Remarks

This extension is useful only when the dump file is a System Crash File (.dmp) that has not been converted to a minidump file from a kernel or full dump file, or the live system has finished starting and is online (for example, at the log-in prompt).

You can use any combination of the **-debug**, **-devices**, **-memory**, **-power**, **-processor**, **-system**, and **-v** parameters in a single extension command.

The following parameters are supported only on particular systems:

- The **gbl** parameter works only when the target computer supports ACPI.
- The **smbios** parameter works only when the target computer supports SMBIOS.
- The **registers** parameter does not work on Itanium-based target computers, because they do not collect MSRs.

Microsoft makes every effort to remove personally identifiable information (PII) from these records. All PII is removed from dump files. However, on a live system, some PII may not yet be removed. As a result, PII fields will be reported as 0 or blank, even if they actually contain information.

To stop execution of commands that include the **cpuinfo**, **gbl**, **registers**, or **smbios** parameters at any time, press CTRL+BREAK (in WinDbg) or CTRL+C (in KD).

!sysptes

1/3/2019 • 2 minutes to read • Edit Online

The **!sysptes** extension displays a formatted view of the system page table entries (PTEs).

```
!sysptes [Flags]
```

Parameters

Flags
Specifies the level of detail to display. *Flags* can be any combination of the following bits. The default is zero:

Bit 0 (0x1)
Displays information about free PTEs.

Bit 1 (0x2)
(Windows 2000 only) Displays unused pages in the page usage statistics.

Displays information about free PTEs in the global special pool.

Bit 2 (0x4)
Displays detailed information about any system PTEs that are allocated to mapping locked pages.

Bit 3 (0x8)
(Windows 2000 and Windows XP only) Displays nonpaged pool expansion free PTE information. If this bit is set, the other lists are not displayed. If both 0x1 and 0x8 are set, all nonpaged pool expansion free PTEs are displayed. If only 0x8 is set, only the total is displayed.

Bit 4 (0x10)
(Windows Vista and later) Displays special pool free PTE information for the session.

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Kdexts.dll

Additional Information

For information about page tables and PTEs, see *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

To examine a specific PTE, use the **!pte** extension.

Here is an example from a Windows system:

```
kd> !sysptes 1

System PTE Information
  Total System Ptes 571224
     SysPtes list of size 1 has 361 free
     SysPtes list of size 2 has 91 free
     SysPtes list of size 4 has 48 free
     SysPtes list of size 8 has 36 free
     SysPtes list of size 9 has 29 free
     SysPtes list of size 23 has 29 free

    starting PTE: fffffe0059388000
    ending PTE:   fffffe00597e3ab8

      free ptes: fffffe0059388000   number free: 551557.
      free ptes: fffffe00597be558   number free: 104.
      free ptes: fffffe00597d2828   number free: 676.

  free blocks: 3   total free: 552337    largest free block: 551557
```

!thread

12/5/2018 • 3 minutes to read • Edit Online

The **!thread** extension displays summary information about a thread on the target system, including the ETHREAD block. This command can be used only during kernel-mode debugging.

This extension command is not the same as the **.thread (Set Register Context)** command.

Syntax

```
!thread [-p] [-t] [Address [Flags]]
```

Parameters

-p
Displays summary information about the process that owns the thread.

-t
When this option is included, *Address* is the thread ID, not the thread address.

Address
Specifies the hexadecimal address of the thread on the target computer. If *Address* is -1 or omitted, it indicates the current thread.

Flags
Specifies the level of detail to display. *Flags* can be any combination of the following bits. If *Flags* is 0, only a minimal amount of information is displayed. The default is 0x6:

Bit 1 (0x2)
Displays the thread's wait states.

Bit 2 (0x4)
If this bit is used without Bit 1 (0x2), it has no effect. If this bit is used with Bit 1, the thread is displayed with a stack trace.

Bit 3 (0x8)
Adds the return address, the stack pointer, and (on Itanium systems) the **bsp** register value to the information displayed for each function and suppresses the display of function arguments.

Bit 4 (0x10)
Sets the process context equal to the process that owns the specified thread for the duration of this command. This results in more accurate display of thread stacks.

DLL

Kdexts.dll

Additional Information

For information about threads in kernel mode, see Changing Contexts. For more information about analyzing processes and threads, see *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

Here is an example using Windows 10:

```
0: kd> !thread 0xffffcb088f0a4480
THREAD ffffcb088f0a4480  Cid 0e34.3814  Teb: 0000001a27ca6000 Win32Thread: 0000000000000000 RUNNING on
processor 0
Not impersonating
DeviceMap                 ffffb80842016c20
Owning Process            ffffcb08905397c0       Image:         MsMpEng.exe
Attached Process          N/A            Image:         N/A
Wait Start TickCount      182835891      Ticks: 0
Context Switch Count      5989           IdealProcessor: 3
UserTime                  00:00:01.046
KernelTime                00:00:00.296
Win32 Start Address 0x00007ffb3b2fd1b0
Stack Init ffff95818476add0 Current ffff958184769d30
Base ffff95818476b000 Limit ffff958184765000 Call 0000000000000000
Priority 8 BasePriority 8 PriorityDecrement 0 IoPriority 2 PagePriority 5
Child-SP          RetAddr           : Args to Child
: Call Site
fffff802`59858c68 fffff801`b56d24aa : ffffcb08`8fd68010 00000000`00000000 fffff802`58259600
00000000`00000008 : nt!DbgBreakPointWithStatus [d:\rs2\minkernel\ntos\rtl\amd64\debugstb.asm @ 130]
fffff802`59858c70 ffffcb08`8fd68010 : 00000000`00000000 fffff802`58259600 00000000`00000008
ffffcb08`8f0a4400 : 0xfffff801`b56d24aa
fffff802`59858c78 00000000`00000000 : fffff802`58259600 00000000`00000008 ffffcb08`8f0a4400
00000000`00000019 : 0xffffcb08`8fd68010
```

Use commands like **!process** to locate the address or thread ID of the thread you are interested in.

The useful information in the **!thread** display is explained in the following table.

PARAMETER	MEANING
Thread address	The hexadecimal number after the word *THREAD* is the address of the ETHREAD block. In the preceding example, the thread address is 0xffffcb088f0a4480 .
Thread ID	The two hexadecimal numbers after the word *Cid* are the process ID and the thread ID: *process ID.thread ID*. In the preceding example, the process ID is 0x0e34, and the thread ID is 0x3814.
Thread Environment Block (TEB)	The hexadecimal number after the word *Teb* is the address of the thread environment block (TEB).
System Service Dispatch Table	The hexadecimal number after the word *Win32Thread* is the address of the system service dispatch table.
Thread State	The thread state is displayed at the end of the line that begins with the word *RUNNING*.
Owning Process	The hexadecimal number after the words *Owning Process* is the address of the EPROCESS for the process that owns this thread.

PARAMETER	MEANING
Start Address	The hexadecimal number after the words *Start Address* is the thread start address. This might appear in symbolic form.
User Thread Function	The hexadecimal number after the words *Win32 Start Address* is the address of the user thread function.
Priority	The priority information for the thread follows the word *Priority*.
Stack trace	A stack trace for the thread appears at the end of this display.

!threadfields

4/8/2019 • 2 minutes to read • Edit Online

The **!threadfields** extension displays the names and offsets of the fields within the executive thread (ETHREAD) block.

```
!threadfields
```

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Unavailable (see the Remarks section)

Additional Information

For information about the ETHREAD block, see *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

This extension command is not available in Windows XP or later versions of Windows. Instead, use the **dt (Display Type)** command to show the ETHREAD structure directly:

```
kd> dt nt!_ETHREAD
```

Here is an example of **!threadfields** from a Windows 2000 system:

```
kd> !threadfields
 ETHREAD structure offsets:

    Tcb:                            0x0
    CreateTime:                     0x1b0
    ExitTime:                       0x1b8
    ExitStatus:                     0x1c0
    PostBlockList:                  0x1c4
    TerminationPortList:            0x1cc
    ActiveTimerListLock:            0x1d4
    ActiveTimerListHead:            0x1d8
    Cid:                            0x1e0
    LpcReplySemaphore:              0x1e8
    LpcReplyMessage:                0x1fc
    LpcReplyMessageId:              0x200
    ImpersonationInfo:              0x208
    IrpList:                        0x20c
    TopLevelIrp:                    0x214
    ReadClusterSize:                0x21c
    ForwardClusterOnly:             0x220
    DisablePageFaultClustering:     0x221
    DeadThread:                     0x222
    HasTerminated:                  0x224
    GrantedAccess:                  0x228
    ThreadsProcess:                 0x22c
    StartAddress:                   0x230
    Win32StartAddress:              0x234
    LpcExitThreadCalled:            0x238
    HardErrorsAreDisabled:          0x239
```

!time

12/5/2018 • 2 minutes to read • Edit Online

The **!time** extension command is obsolete. Use the **.time (Display System Time)** command instead.

!timer

12/5/2018 • 2 minutes to read • Edit Online

The **!timer** extension displays a detailed listing of all system timer use.

```
!timer
```

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Kdexts.dll

Additional Information

For information about timer objects, see the Windows Driver Kit (WDK) documentation.

Remarks

The **!timer** extension displays the timer tree, which stores all timer objects in the system.

Here is an example:

```
kd> !timer
Dump system timers

Interrupt time: 9f760774 00000000 [12/ 8/2000 10:59:22.685 (Pacific Standard Time)]

List Timer    Interrupt Low/High     Fire Time              DPC/thread
 0 8016aea0 P 9fbd8e00 00000000 [12/ 8/2000 10:59:23.154]  ntoskrnl!PopScanIdleList
  1 8257f118   e4e4225a 00000000 [12/ 8/2000 11:01:19.170]  thread 8257f030
 3 80165fc0   286be1c9 0000594a [ 4/ 1/2001 01:59:59.215]  ntoskrnl!ExpTimeZoneDpcRoutine
    80165f40   2a7bf8d9 006f105e [12/31/2099 23:59:59.216]  ntoskrnl!ExpCenturyDpcRoutine
  5 825a0bf8   a952e1c2 00000000 [12/ 8/2000 10:59:39.232]  thread 825a0b10
 10 8251c7a8   41f54d84 00000001 [12/ 8/2000 11:03:55.310]  thread 8251c6c0
    8249fe88   41f54d84 00000001 [12/ 8/2000 11:03:55.310]  thread 8249fda0
 11 8250e7e8   bc73ffde 00000000 [12/ 8/2000 11:00:11.326]  thread 8250e700

.....

237 82757070   9f904152 00000000 [12/ 8/2000 10:59:22.857]  +f7a56f2e
    82676348   9f904152 00000000 [12/ 8/2000 10:59:22.857]  +fe516352
    82728b78   9f904152 00000000 [12/ 8/2000 10:59:22.857]  +fe516352
238 fe4b5d78   9f92a3ac 00000000 [12/ 8/2000 10:59:22.873]  thread 827ceb10
    801658f0   9f92a3ac 00000000 [12/ 8/2000 10:59:22.873]  ntoskrnl!CcScanDpc
239 8259ad40   765a6f19 00000bba [12/23/2000 09:07:22.900]  thread 825d3670
250 826d42f0   1486bed8 80000000 [         NEVER         ]  thread 825fa030

Total Timers: 193, Maximum List: 7
Current Hand: 226, Maximum Search: 0

Wakeable timers:
```

!tokenfields

12/5/2018 • 2 minutes to read • Edit Online

The **!tokenfields** extension displays the names and offsets of the fields within the access token object (the TOKEN structure).

```
!tokenfields
```

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Unavailable (see the Remarks section)

Additional Information

For information about the TOKEN structure, see *Microsoft Windows Internals*, by Mark Russinovich and David Solomon. This book may not be available in some languages and countries.(The user-mode token structures described in the Microsoft Windows SDK documentation are slightly different.)

Remarks

This extension command is not available in Windows XP or later versions of Windows. Instead, use the **dt (Display Type)** command to show the TOKEN structure directly:

```
kd> dt nt!_TOKEN
```

To see a specific instance of the TOKEN structure, use the **!token** extension.

Here is an example of **!tokenfields** from a Windows 2000 system:

```
kd> !tokenfields
 TOKEN structure offsets:
    TokenSource:           0x0
    AuthenticationId:      0x18
    ExpirationTime:        0x28
    ModifiedId:            0x30
    UserAndGroupCount:     0x3c
    PrivilegeCount:        0x44
    VariableLength:        0x48
    DynamicCharged:        0x4c
    DynamicAvailable:      0x50
    DefaultOwnerIndex:     0x54
    DefaultDacl:           0x6c
    TokenType:             0x70
    ImpersonationLevel:    0x74
    TokenFlags:            0x78
    TokenInUse:            0x79
 ProxyData:             0x7c
    AuditData:             0x80
    VariablePart:          0x84
```

!trap

12/5/2018 • 2 minutes to read • Edit Online

The **!trap** extension command is obsolete. Use the **.trap (Display Trap Frame)** command instead.

!tss

12/5/2018 • 2 minutes to read • Edit Online

The **!tss** extension command is obsolete. Use the **.tss (Display Task State Segment)** command instead.

!tz

8/26/2019 • 2 minutes to read • Edit Online

The **!tz** extension displays the specified power thermal zone structure.

```
!tz [Address]
```

Parameters

Address
The address of a power thermal zone that you want to display. If this parameter is omitted, the display includes all thermal zones on the target computer.

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Kdexts.dll

Additional Information

To view the system's power capabilities, use the **!pocaps** extension command. To view the system's power policy, use the **!popolicy** extension command. For information about power capabilities and power policy, see the Windows Driver Kit (WDK) documentation and *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

To stop execution at any time, press CTRL+BREAK (in WinDbg) or CTRL+C (in KD).

!tzinfo

8/26/2019 • 2 minutes to read • Edit Online

The **!tzinfo** extension displays the contents of the specified thermal zone information structure.

```
!tzinfo Address
```

Parameters

Address
The address of a thermal zone information structure that you want to display.

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Kdexts.dll

Additional Information

To view the system's power capabilities, use the **!pocaps** extension command. To view the system's power policy, use the **!popolicy** extension command. For information about power capabilities and power policy, see the Windows Driver Kit (WDK) documentation and *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

!ubc

12/5/2018 • 2 minutes to read • Edit Online

The **!ubc** extension clears a user-space breakpoint.

```
!ubc BreakpointNumber
```

Parameters

BreakpointNumber
Specifies the number of the breakpoint to be cleared. An asterisk (*) indicates all breakpoints.

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Kdexts.dll

Remarks

This will permanently delete a breakpoint set with **!ubp**.

See also

!ubd

!ube

!ubl

!ubp

User Space and System Space

!ubd

12/5/2018 • 2 minutes to read • Edit Online

The **!ubd** extension temporarily disables a user-space breakpoint.

```
!ubd BreakpointNumber
```

Parameters

BreakpointNumber
Specifies the number of the breakpoint to be disabled. An asterisk (*) indicates all breakpoints.

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Kdexts.dll

Remarks

Disabled breakpoints will be ignored. Use **!ube** to re-enable the breakpoint.

See also

!ubc

!ube

!ubl

!ubp

User Space and System Space

!ube

12/5/2018 • 2 minutes to read • Edit Online

The **!ube** extension re-enables a user-space breakpoint.

```
!ube BreakpointNumber
```

Parameters

BreakpointNumber
Specifies the number of the breakpoint to be enabled. An asterisk (*) indicates all breakpoints.

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Kdexts.dll

Remarks

This is used to re-enable a breakpoint that was disabled by **!ubd**.

See also

!ubc

!ubd

!ubl

!ubp

User Space and System Space

!ubl

12/5/2018 • 2 minutes to read • Edit Online

The **!ubl** extension lists all user-space breakpoints and their current status.

```
!ubl
```

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Kdexts.dll

Remarks

Here is an example of the use and display of user-space breakpoints:

```
kd> !ubp 8014a131
This command is VERY DANGEROUS, and may crash your system!
If you don't know what you are doing, enter "!ubc *" now!

kd> !ubp 801544f4

kd> !ubd 1

kd> !ubl
 0: e ffffffff`8014a131 (ffffffff`82deb000) 1 ffffffff
 1: d ffffffff`801544f4 (ffffffff`82dff000) 0 ffffffff
```

Each line in this listing contains the breakpoint number, the status (**e** for enabled or **d** for disabled), the virtual address used to set the breakpoint, the physical address of the actual breakpoint, the byte position, and the contents of this memory location at the time the breakpoint was set.

See also

!ubc

!ubd

!ube

!ubp

User Space and System Space

!ubp

12/5/2018 • 2 minutes to read • Edit Online

The **!ubp** extension sets a breakpoint in user space.

```
!ubp Address
```

Parameters

Address
Specifies the hexadecimal virtual address of the location in user space where the breakpoint is to be set.

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Kdexts.dll

Remarks

The **!ubp** extension sets a breakpoint in user space. The breakpoint is set on the actual physical page, not just the virtual page.

Setting a physical breakpoint will simultaneously modify every virtual copy of a page, with unpredictable results. One possible consequence is corruption of the system state, possibly followed by a bug check or other system crash. Therefore, these breakpoints should be used cautiously, if at all.

This extension cannot be used to set breakpoints on pages that have been swapped out of memory. If a page is swapped out of memory after a breakpoint is set, the breakpoint ceases to exist.

It is not possible to set a breakpoint inside a page table or a page directory.

Each breakpoint is assigned a *breakpoint number*. To find out the breakpoint number assigned, use **!ubl**. Breakpoints are enabled upon creation. To step over a breakpoint, you must first disable it by using **!ubd**. To clear a breakpoint, use **!ubc**.

See also

!ubc

!ubd

!ube

!ubl

User Space and System Space

!urb

12/5/2018 • 2 minutes to read • Edit Online

The **!urb** extension command is obsolete. Use the **dt URB** command instead.

!vad

4/12/2019 • 2 minutes to read • Edit Online

The **!vad** extension displays details of a virtual address descriptor (VAD) or a tree of VADs.

- Displays details of one virtual address descriptor (VAD)
- Displays details of a tree of VADs.
- Displays information about the VADs for a particular user-mode module and provides a string that you can use to load the symbols for that module.

```
!vad VAD-Root [Flag]
!vad Address 1
```

Parameters

VAD-Root
Address of the root of the VAD tree to be displayed.

Flag
Specifies the form the display will take. Possible values include:

0
The entire VAD tree based at *VAD-Root* is displayed. (This is the default.)

1
Only the VAD specified by *VAD-Root* is displayed. The display will include a more detailed analysis.

Address
Address in the virtual address range of a user-mode module.

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Kdexts.dll

Additional Information

For information about virtual address descriptors, see *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

The address of the root of the VAD for any process can be found by using the **!process** command.

The **!vad** command can be helpful when you need to load symbols for a user-mode module that has been paged out of memory. For details, see Mapping Symbols When the PEB is Paged Out.

Here is an example of the **!vad** extension:

```
kd> !vad 824bc2f8
VAD     level      start      end    commit
82741bf8 ( 1)      78000    78045         8 Mapped  Exe  EXECUTE_WRITECOPY
824ef368 ( 2)      7f6f0    7f7ef         0 Mapped       EXECUTE_READ
824bc2f8 ( 0)      7ffb0    7ffd3         0 Mapped       READONLY
8273e508 ( 2)      7ffde    7ffde         1 Private      EXECUTE_READWRITE
82643fc8 ( 1)      7ffdf    7ffdf         1 Private      EXECUTE_READWRITE

Total VADs:     5  average level:    2  maximum depth: 2

kd> !vad 824bc2f8 1

VAD @ 824bc2f8
  Start VPN:         7ffb0  End VPN:    7ffd3  Control Area:  827f1208
  First ProtoPte: e1008500  Last PTE e100858c  Commit Charge         0 (0.)
  Secured.Flink          0  Blink           0  Banked/Extend:        0 Offset 0
   ViewShare NoChange READONLY

SecNoChange
```

!vad_reload

4/8/2019 • 2 minutes to read • Edit Online

The **!vad_reload** extension reloads the user-mode modules for a specified process by using the virtual address descriptors (VADs) of that process.

```
!vad_reload Process
```

Parameters

Process
Specifies the hexadecimal address of the process for which the modules will be loaded.

Additional Information

For information about VADs, see *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

You can use the **!process** extension to find the address of a process.

Here is an example of how to find an address and use it in the **!vad_reload** command.

```
3: kd> !process 0 0
. . .
PROCESS fffffa8007d54450
    SessionId: 1  Cid: 115c    Peb: 7ffffef6000  ParentCid: 0c58
    DirBase: 111bc3000  ObjectTable: fffff8a006ae0960  HandleCount: 229.
    Image: SearchProtocolHost.exe
. . .
3: kd> !vad_reload fffffa8007d54450
fffffa80`04f5e8b0: VAD maps 00000000`6c230000 - 00000000`6c26bfff, file cscobj.dll
fffffa80`04e8f890: VAD maps 00000000`6ef90000 - 00000000`6f04afff, file mssvp.dll
fffffa80`07cbb010: VAD maps 00000000`6f910000 - 00000000`6faf5fff, file tquery.dll
fffffa80`08c1f2a0: VAD maps 00000000`6fb80000 - 00000000`6fb9bfff, file mssprxy.dll
fffffa80`07dce8b0: VAD maps 00000000`6fba0000 - 00000000`6fba7fff, file msshooks.dll
fffffa80`04fd2e70: VAD maps 00000000`72a50000 - 00000000`72a6cfff, file userenv.dll
. . .
```

Requirements

DLL

Kdexts.dll

See also

!process

!vad

!validatelist

12/5/2018 • 2 minutes to read • Edit Online

The **!validatelist** extension verifies that the backward and forward links in a doubly-linked list are valid.

```
!validatelist Address
```

Parameters

Address
The address of the doubly-linked list.

DLL

Windows 2000	Unavailable
Windows XP and later	Kdexts.dll

Remarks

To stop execution, press Ctrl+Break (in WinDbg) or Ctrl+C (in KD).

!verifier

6/25/2019 • 8 minutes to read • Edit Online

The **!verifier** extension displays the status of Driver Verifier and its actions.

Driver Verifier is included in Windows. It works on both checked and free builds. For information about Driver Verifier, see the Driver Verifier topic in the Windows Driver Kit (WDK) documentation.

Syntax

```
!verifier [Flags [Image]]
!verifier 4 [Quantity]
!verifier 8 [Quantity]
!verifier 0x40 [Quantity]
!verifier 0x80 [Quantity]
!verifier 0x80 Address
!verifier 0x100 [Quantity]
!verifier 0x100 Address
!verifier 0x200 [Address]
!verifier 0x400 [Address]
!verifier -disable
!verifier ?
```

Parameters

Flags
Specifies what information is displayed in the output from this command. If *Flags* is equal to the value 4, 8, 0x20, 0x40, 0x80, or 0x100, then the remaining arguments to **!verifier** are interpreted based on the specific arguments associated with those values. If *Flags* is equal to any other value, even if one or more of these bits are set, only the *Flags* and *Image* arguments are permitted. *Flags* can be any sum of the following bits; the default is 0:

Bit 0 (0x1)
Displays the names of all drivers being verified. The number of bytes currently allocated to each driver from the nonpaged pool and the paged pool is also displayed.

Bit 1 (0x2)
Displays information about pools (pool size, headers, and pool tags) and outstanding memory allocations left by unloaded drivers. This flag has no effect unless bit 0 (0x1) is also set.

Bit 2 (0x4)
Displays fault injection information. The return address, symbol name, and displacement of the code requesting each allocation are displayed. If *Flags* is exactly 0x4 and the *Quantity* parameter is included, the number of these records displayed can be chosen. Otherwise, four records are displayed.

Bit 3 (0x8)
Displays the most recent IRQL changes made by the drivers being verified. The old IRQL, new IRQL, processor, and time stamp are displayed. If *Flags* is exactly 0x8 and the *Quantity* parameter is included, the number of these records displayed can be chosen. Otherwise, four records are displayed.

Warning In 64-bit versions of Windows, some of the kernel functions that raise or lower the IRQL are implemented as inline code rather than as exported functions. Driver Verifier does not report IRQL changes made by inline code, so it is possible for the IRQL transition log produced by Driver Verifier to be incomplete. See Remarks for an example of a missing IRQL transition entry.

Bit 6 (0x40)
(Windows Vista and later) Displays information from the **Force Pending I/O Requests** option of Driver Verifier, including traces from the log of forced pending IRPs.

The *Quantity* parameter specifies the number of traces to be displayed. By default, the entire log is displayed.

Bit 7 (0x80)
(Windows Vista and later) Displays information from the kernel pool Allocate/Free log.

The *Quantity* parameter specifies the number of traces to be displayed. By default, the entire log is displayed.

If *Address* is specified, only traces associated with the specified address within the kernel pool Allocate/Free log are displayed.

Bit 8 (0x100)
(Windows Vista and later) Displays information from the log of IoAllocateIrp, IoCompleteRequest and IoCancelIrp calls.

The Quantity parameter specifies the number of traces to be displayed. By default, the entire log is displayed.

If *Address* is specified, only traces associated with the specified IRP address are displayed.

Bit 9 (0x200)
(Windows Vista and later) Displays entries in the Critical Region log.

If *Address* is specified, only entries associated with the specified thread address are displayed.

Bit 10 (0x400)
(Windows Vista and later) Displays cancelled IRPs that are currently being watched by Driver Verifier.

If *Address* is specified, only the IRP with the specified address is displayed.

Bit 11 (0x800)
(Windows 8.1 and later) Display entries from the fault injection log that is created when you select the Systematic low resource simulation option.

Image
If *Flags* is used and is not equal to 4, 8, or 0x10, *Image* specifies the name of a driver. *Image* is used to filter the information displayed by *Flags* values of 0x1 and 0x2: only the specified driver is considered. This driver must be currently verified.

Quantity
If *Flags* is exactly equal to 0x4, *Quantity* specifies the number of fault injection records to display. If *Flags* is exactly equal to 0x8, *Quantity* specifies the number of IRQL log entries to display. If *Flags* is exactly equal to 0x40, *Quantity* specifies the number of traces displayed from the log of forced pending IRPs. If Flags is exactly equal to 0x80, Quantity specifies the number of traces displayed from the kernel pool Allocate/Free log. If Flags is exactly equal to 0x100, Quantity specifies the number of traces displayed from the log of IoAllocateIrp, IoCompleteRequest and IoCancelIrp calls.

-disable
Clears the current Driver Verifier settings on the debug target. The clearing of these settings does not persist through a reboot. If you need to disable the Driver Verifier settings to successfully boot, set a breakpoint at nt!VerifierInitSystem and use the **!verifier -disable** command at that point.

?
Displays some brief Help text for this extension in the Debugger Command window.

DLL

Kdexts.dll

Additional Information

For information about Driver Verifier, see the Windows Driver Kit (WDK) documentation.

Remarks

The following example illustrates that on 64-bit versions of Windows, the IRQL transition log is not always complete. The two entries shown are consecutive entries in the log for Processor 2. The first entry shows the IRQL going from 2 to 0. The second entry shows the IRQL going from 2 to 2. Information about how the IRQL got raised from 0 to 2 is missing.

```
Thread:             fffffa80068c9400
Old irql:           0000000000000002
New irql:           0000000000000000
Processor:          0000000000000002
Time stamp:         0000000000000857

    fffff8800140f12a ndis!ndisNsiGetInterfaceInformation+0x20a
    fffff88001509478 NETIO!NsiGetParameterEx+0x178
    fffff88005f062f2 nsiproxy!NsippGetParameter+0x24a
    fffff88005f086db nsiproxy!NsippDispatchDeviceControl+0xa3
    fffff88005f087a0 nsiproxy!NsippDispatch+0x48

Thread:             fffffa80068c9400
Old irql:           0000000000000002
New irql:           0000000000000002
Processor:          0000000000000002
Time stamp:         0000000000000857

    fffff8800140d48d ndis!ndisReferenceTopMiniportByNameForNsi+0x1ce
    fffff8800140f072 ndis!ndisNsiGetInterfaceInformation+0x152
    fffff88001509478 NETIO!NsiGetParameterEx+0x178
    fffff88005f062f2 nsiproxy!NsippGetParameter+0x24a
    fffff88005f086db nsiproxy!NsippDispatchDeviceControl+0xa3
```

When using Driver Verifier to test graphics drivers, use the **!gdikdx.verifier** extension instead of **!verifier**.

The values of 4, 8, and 0x20, 0x40, 0x80, and 0x100 are special values for *Flags*. If these values are used, the special arguments listed in the **Parameters** section can be used, and the display will include only the information associated with that flag value.

If any other value for *Flags* is used, even if one or more of these bits are set, only the *Flags* and *Image* arguments are permitted. In this situation, in addition to all the other information displayed, **!verifier** will display the Driver Verifier options that are active, along with statistics on pool allocations, IRQL raises, spin locks, and trims.

If *Flags* equals 0x20, the values specified for *CompletionTime*, *CancelTime*, and *ForceCancellation* are used by the Driver Hang Verification option of Driver Verifier. These new values take effect immediately and last until the next boot. When you reboot, they revert to their default values.

Also, if *Flags* equals 0x20 (with or without additional parameters), the Driver Hang Verification log is printed. For information on interpreting the log, see the Driver Hang Verification section of the Driver Verifier documentation in the Windows Driver Kit (WDK) documentation.

Here is an example of the **!verifier** extension on a Windows 7 computer.

```
2: kd> !verifier 0xf

Verify Level 9bb ... enabled options are:
    Special pool
    Special irql
    All pool allocations checked on unload
    Io subsystem checking enabled
```

```
    Deadlock detection enabled
    DMA checking enabled
    Security checks enabled
    Miscellaneous checks enabled

Summary of All Verifier Statistics

RaiseIrqls                          0x0
AcquireSpinLocks                    0x362
Synch Executions                    0x0
Trims                               0xa34a

Pool Allocations Attempted          0x7b058
Pool Allocations Succeeded          0x7b058
Pool Allocations Succeeded SpecialPool 0x7b058
Pool Allocations With NO TAG        0x0
Pool Allocations Failed             0x0
Resource Allocations Failed Deliberately   0x0

Current paged pool allocations      0x1a for 00000950 bytes
Peak paged pool allocations         0x1b for 00000AC4 bytes
Current nonpaged pool allocations   0xe3 for 00046110 bytes
Peak nonpaged pool allocations      0x10f for 00048E40 bytes

Driver Verification List

Entry     State           NonPagedPool   PagedPool   Module

fffffa8003b6f670 Loaded           000000a0       00000854    videoprt.sys

Current Pool Allocations  00000002    00000013
Current Pool Bytes        000000a0    00000854
Peak Pool Allocations     00000006    00000014
Peak Pool Bytes           000008c0    000009c8

PoolAddress  SizeInBytes    Tag       CallersAddress
fffff9800157efc0     0x0000003c     Vprt      fffff88002c62963
fffff9800146afc0     0x00000034     Vprt      fffff88002c62963
fffff980015bafe0     0x00000018     Vprt      fffff88002c628f7
...

fffffa8003b6f620 Loaded           00046070       000000fc    usbport.sys

Current Pool Allocations  000000e1    00000007
Current Pool Bytes        00046070    000000fc
Peak Pool Allocations     0000010d    0000000a
Peak Pool Bytes           00048da0    00000254

PoolAddress  SizeInBytes    Tag       CallersAddress
fffff98003a38fc0     0x00000038     usbp      fffff88004215e34
fffff98003a2cfc0     0x00000038     usbp      fffff88004215e34
fffff9800415efc0     0x00000038     usbp      fffff88004215e34
...

------------------------------------------------
Fault injection trace log
------------------------------------------------

Driver Verifier didn't inject any faults.

------------------------------------------------
Track irql trace log
------------------------------------------------

Displaying most recent 0x0000000000000004 entries from the IRQL transition log.
There are up to 0x100 entries in the log.

Thread:             fffff80002bf8c40
Old irql:           0000000000000002
```

```
Old irql:           0000000000000002
New irql:           0000000000000002
Processor:          0000000000000000
Time stamp:         000000000000495e

    fffff8800420f2ca USBPORT!USBPORT_DM_IoTimerDpc+0x9a
    fffff80002a5b5bf nt!IopTimerDispatch+0x132
    fffff80002a7c29e nt!KiProcessTimerDpcTable+0x66
    fffff80002a7bdd6 nt!KiProcessExpiredTimerList+0xc6
    fffff80002a7c4be nt!KiTimerExpiration+0x1be

Thread:             fffff80002bf8c40
Old irql:           0000000000000002
New irql:           0000000000000002
Processor:          0000000000000000
Time stamp:         000000000000495e

    fffff88004205f3a USBPORT!USBPORT_AcquireEpListLock+0x2e
    fffff880042172df USBPORT!USBPORT_Core_TimeoutAllTransfers+0x1f
    fffff8800420f2ca USBPORT!USBPORT_DM_IoTimerDpc+0x9a
    fffff80002a5b5bf nt!IopTimerDispatch+0x132
    fffff80002a7c29e nt!KiProcessTimerDpcTable+0x66

Thread:             fffff80002bf8c40
Old irql:           0000000000000002
New irql:           0000000000000002
Processor:          0000000000000000
Time stamp:         000000000000495e

    fffff88004201694 USBPORT!MPf_CheckController+0x4c
    fffff8800420f26a USBPORT!USBPORT_DM_IoTimerDpc+0x3a
    fffff80002a5b5bf nt!IopTimerDispatch+0x132
    fffff80002a7c29e nt!KiProcessTimerDpcTable+0x66
    fffff80002a7bdd6 nt!KiProcessExpiredTimerList+0xc6

Thread:             fffff80002bf8c40
Old irql:           0000000000000002
New irql:           0000000000000002
Processor:          0000000000000000
Time stamp:         000000000000495e

    fffff8800420167c USBPORT!MPf_CheckController+0x34
    fffff8800420f26a USBPORT!USBPORT_DM_IoTimerDpc+0x3a
    fffff80002a5b5bf nt!IopTimerDispatch+0x132
    fffff80002a7c29e nt!KiProcessTimerDpcTable+0x66
    fffff80002a7bdd6 nt!KiProcessExpiredTimerList+0xc6
```

Here is an example of the **!verifier** extension on a Windows Vista computer with bit 7 turned on and *Address* specified.

```
0: kd> !verifier 80 a2b1cf20
# Parsing 00004000 array entries, searching for address a2b1cf20.

Pool block a2b1ce98, Size 00000168, Thread a2b1ce98
808f1be6 ndis!ndisFreeToNPagedPool+0x39
808f11c1 ndis!ndisPplFree+0x47
808f100f ndis!NdisFreeNetBufferList+0x3b
8088db41 NETIO!NetioFreeNetBufferAndNetBufferList+0xe
8c588d68 tcpip!UdpEndSendMessages+0xdf
8c588cb5 tcpip!UdpSendMessagesDatagramsComplete+0x22
8088d622 NETIO!NetioDereferenceNetBufferListChain+0xcf
8c5954ea tcpip!FlSendNetBufferListChainComplete+0x1c
809b2370 ndis!ndisMSendCompleteNetBufferListsInternal+0x67
808f1781 ndis!NdisFSendNetBufferListsComplete+0x1a
8c04c68e pacer!PcFilterSendNetBufferListsComplete+0xb2
809b230c ndis!NdisMSendNetBufferListsComplete+0x70
# 8ac4a8ba test1!HandleCompletedTxPacket+0xea

Pool block a2b1ce98, Size 00000164, Thread a2b1ce98
822af87f nt!VerifierExAllocatePoolWithTagPriority+0x5d
808f1c88 ndis!ndisAllocateFromNPagedPool+0x1d
808f11f3 ndis!ndisPplAllocate+0x60
808f1257 ndis!NdisAllocateNetBufferList+0x26
80890933 NETIO!NetioAllocateAndReferenceNetBufferListNetBufferMdlAndData+0x14
8c5889c2 tcpip!UdpSendMessages+0x503
8c05c565 afd!AfdTLSendMessages+0x27
8c07a087 afd!AfdTLFastDgramSend+0x7d
8c079f82 afd!AfdFastDatagramSend+0x5ae
8c06f3ea afd!AfdFastIoDeviceControl+0x3c1
8217474f nt!IopXxxControlFile+0x268
821797a1 nt!NtDeviceIoControlFile+0x2a
8204d16a nt!KiFastCallEntry+0x127
```

!vm

8/26/2019 • 2 minutes to read • Edit Online

The **!vm** extension displays summary information about virtual memory use statistics on the target system.

```
!vm [Flags]
```

Parameters

Flags
Specifies what information will be displayed in the output from this command. This can be any sum of the following bits. The default is 0, which causes the display to include system-wide virtual memory statistics as well as memory statistics for each process.

Bit 0 (0x1)
Causes the display to omit process-specific statistics.

Bit 1 (0x2)
Causes the display to include memory management thread stacks.

Bit 2 (0x4)
Causes the display to include terminal server memory usage.

Bit 3 (0x8)
Causes the display to include the page file write log.

Bit 4 (0x10)
Causes the display to include working set owner thread stacks.

Bit 5 (0x20)
(Windows Vista and later) Causes the display to include kernel virtual address usage.

Environment

Modes	kernel mode only

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Kdexts.dll

Additional Information

The **!memusage** extension command can be used to analyze physical memory usage. For more information about memory management, see *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

Here is an example of the short output produced when *Flags* is 1:

```
kd> !vm 1

*** Virtual Memory Usage ***
      Physical Memory:     16270   (   65080 Kb)
      Page File: \??\E:\pagefile.sys
         Current:     98304Kb Free Space:     61044Kb
 Minimum:     98304Kb Maximum:       196608Kb
      Available Pages:      5543   (   22172 Kb)
      ResAvail Pages:       6759   (   27036 Kb)
      Locked IO Pages:       112   (     448 Kb)
 Free System PTEs:    45089   (  180356 Kb)
      Free NP PTEs:         5145   (   20580 Kb)
      Free Special NP:       336   (    1344 Kb)
      Modified Pages:        714   (    2856 Kb)
      NonPagedPool Usage:    877   (    3508 Kb)
      NonPagedPool Max:     6252   (   25008 Kb)
      PagedPool 0 Usage:     729   (    2916 Kb)
      PagedPool 1 Usage:     432   (    1728 Kb)
      PagedPool 2 Usage:     436   (    1744 Kb)
      PagedPool Usage:      1597   (    6388 Kb)
      PagedPool Maximum:   13312   (   53248 Kb)
      Shared Commit:        1097   (    4388 Kb)
      Special Pool:          229   (     916 Kb)
      Shared Process:       1956   (    7824 Kb)
      PagedPool Commit:     1597   (    6388 Kb)
      Driver Commit:         828   (    3312 Kb)
      Committed pages:     21949   (   87796 Kb)
      Commit limit:        36256   (  145024 Kb)
```

All memory use is listed in pages and in kilobytes. The most useful information in this display is the following:

PARAMETER	MEANING
physical memory	Total physical memory in the system.
available pages	Number of pages of memory available on the system, both virtual and physical.
nonpaged pool usage	The amount of pages allocated to the nonpaged pool. The nonpaged pool is memory that cannot be swapped out to the paging file, so it must always occupy physical memory. If this number is too large, this is usually an indication that there is a memory leak somewhere in the system.

!vpb

8/26/2019 • 2 minutes to read • Edit Online

The **!vpb** extension displays a volume parameter block (VPB).

```
!vpb Address
```

Parameters

Address
Specifies the hexadecimal address of the VPB.

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Kdexts.dll

Additional Information

For information about VPBs, see the Windows Driver Kit (WDK) documentation and *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

Here is an example. First, the device tree is displayed with the **!devnode** extension:

```
kd> !devnode 0 1
Dumping IopRootDeviceNode (= 0x80e203b8)
DevNode 0x80e203b8 for PDO 0x80e204f8
  InstancePath is "HTREE\ROOT\0"
  State = DeviceNodeStarted (0x308)
  Previous State = DeviceNodeEnumerateCompletion (0x30d)
  DevNode 0x80e56dc8 for PDO 0x80e56f18
    InstancePath is "Root\dmio\0000"
    ServiceName is "dmio"
    State = DeviceNodeStarted (0x308)
    Previous State = DeviceNodeEnumerateCompletion (0x30d)
  DevNode 0x80e56ae8 for PDO 0x80e56c38
    InstancePath is "Root\ftdisk\0000"
    ServiceName is "ftdisk"
    State = DeviceNodeStarted (0x308)
    Previous State = DeviceNodeEnumerateCompletion (0x30d)
 DevNode 0x80e152a0 for PDO 0x80e15cb8
      InstancePath is "STORAGE\Volume\1&30a96598&0&Signature5C34D70COffset7E00Length60170A00"
      ServiceName is "VolSnap"
      TargetDeviceNotify List - f 0xe1250938  b 0xe14b9198
      State = DeviceNodeStarted (0x308)
      Previous State = DeviceNodeEnumerateCompletion (0x30d)
    .....
```

The last device node listed is a volume. Examine its physical device object (PDO) with the **!devobj** extension:

```
kd> !devobj 80e15cb8
Device object (80e15cb8) is for:
 HarddiskVolume1 \Driver\Ftdisk DriverObject 80e4e248
Current Irp 00000000 RefCount 14 Type 00000007 Flags 00001050
Vpb 80e15c30 DevExt 80e15d70 DevObjExt 80e15e40 Dope 80e15bd8 DevNode 80e152a0
ExtensionFlags (0000000000)
AttachedDevice (Upper) 80e14c60 \Driver\VolSnap
Device queue is not busy.
```

The address of this device's VPB is included in this listing. Use this address with the **!vpb** extension:

```
kd> !vpb 80e15c30
Vpb at 0x80e15c30
Flags: 0x1 mounted
DeviceObject: 0x80de5020
RealDevice:   0x80e15cb8
RefCount: 14
Volume Label:           MY-DISK-C
```

!vpdd

12/5/2018 • 2 minutes to read • Edit Online

The **!vpdd** extension is obsolete. Use **!vtop** or **!ptov** instead.

!vtop

12/5/2018 • 2 minutes to read • Edit Online

The **!vtop** extension converts a virtual address to the corresponding physical address, and displays other page table and page directory information.

Syntax

```
!vtop PFN VirtualAddress
!vtop 0 VirtualAddress
```

Parameters

DirBase
Specifies the directory base for the process. Each process has its own virtual address space. Use the **!process** extension to determine the directory base for a process.

PFN
Specifies the page frame number (PFN) of the directory base for the process.

0
Causes **!vtop** to use the current process context for address translation.

VirtualAddress
Specifies the virtual address whose page is desired.

DLL

Kdexts.dll

Additional Information

For other methods of achieving these results, see Converting Virtual Addresses to Physical Addresses. Also see **!ptov**. For information about page tables and page directories, see *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

To use this command, first use the **!process** extension to determine the directory base of the process. The page frame number (PFN) of this directory base can be found by removing the three trailing hexadecimal zeros (in other words, by right-shifting the number 12 bits).

Here is an example:

```
kd> !process 0 0
**** NT ACTIVE PROCESS DUMP ****
....
PROCESS ff779190  SessionId: 0  Cid: 04fc    Peb: 7ffdf000  ParentCid: 0394
 DirBase: 098fd000  ObjectTable: e1646b30  TableSize:   8.
    Image: MyApp.exe
```

Since the directory base is 0x098FD000, its PFN is 0x098FD.

```
kd> !vtop 98fd 12f980
Pdi 0 Pti 12f
0012f980 09de9000 pfn(09de9)
```

Notice how the trailing three zeros are optional. The **!vtop** extension displays the page directory index (PDI), the page table index (PTI), the virtual address that you had originally input, the physical address of the beginning of the physical page, and the page frame number (PFN) of the page table entry (PTE).

If you want to convert the virtual address 0x0012F980 to a physical address, you simply have to take the last three hexadecimal digits (0x980) and add them to the physical address of the beginning of the page (0x09DE9000). This gives the physical address 0x09DE9980.

If you forget to remove the three zeros, and pass the full directory base to **!vtop** instead of the PFN, the results will usually be correct. This is because when **!vtop** receives a number too large to be a PFN, it right-shifts it twelve bits and uses that number instead:

```
kd> !vtop 98fd 12f980
Pdi 0 Pti 12f
0012f980 09de9000 pfn(09de9)

kd> !vtop 98fd000 12f980
Pdi 0 Pti 12f
0012f980 09de9000 pfn(09de9)
```

However, it is better to always use the PFN, because some directory base values will not be converted in this manner.

!wdmaud

12/5/2018 • 2 minutes to read • Edit Online

Displays a variety of WDM Audio (WDMAud) structures.

```
!wdmaud Address Flags
```

Parameters

Address
Specifies the address of the structure to be displayed.

Flags
Specifies the information to display. This must include exactly one of the bits 0x1, 0x2, 0x4, and 0x8. The 0x100 bit can be added to any of these.

Bit 0 (0x1)
Displays a list of all IOCTLs that have been sent to wdmaud.sys. When this is used, *Address* should specify the address of the **WdmaIoctlHistoryListHead**. If the 0x100 bit is set, the display also includes the **pContext** that each IOCTL was sent with.

Bit 1 (0x2)
Displays a list of all IRPs that WDMAud has marked as pending. When this is used, *Address* should specify the address of the **WdmaPendingIrpListHead**. If the 0x100 bit is set, the display also includes the context on which each IRP was allocated.

Bit 2 (0x4)
Displays a list of all MDLs that WDMAud has allocated. When this is used, *Address* should specify the address of the **WdmaAllocatedMdlListHead**. If the 0x100 bit is set, the display also includes the context on which each MDL was allocated.

Bit 3 (0x8)
Displays a list of all active contexts attached to wdmaud.sys. When this is used, *Address* should specify the address of the **WdmaContextListHead**. If the 0x100 bit is set, the display also includes the data members of each context structure.

Bit 8 (0x100)
Causes the display to include verbose information.

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Unavailable

Additional Information

For information about WDM audio architecture and audio drivers, see the Windows Driver Kit (WDK) documentation.

Remarks

The contexts attached to wdmaud.sys (**pContext**) contain most of the state data for each device. Whenever wdmaud.drv is loaded into a new process, wdmaud.sys is notified of its arrival. Whenever wdmaud.drv is unloaded, wdmaud.sys cleans up any allocations made in that context.

!whattime

12/5/2018 • 2 minutes to read • Edit Online

The **!whattime** extension converts a tick count into a standard time value.

```
!whattime Ticks
```

Parameters

Ticks
The number of ticks.

DLL

Windows 2000	Unavailable
Windows XP and later	Kdexts.dll

Remarks

The output is displayed as *HH:MM:SS.mmm*. Here is an example:

```
kd> !whattime 29857ae4
696613604 Ticks in Standard Time:  15:02:16.040s
```

!whatperftime

12/5/2018 • 2 minutes to read • Edit Online

The **!whatperftime** extension converts a high-resolution performance counter value into a standard time value.

```
!whatperftime Count
```

Parameters

Count
The performance counter clock value.

DLL

Windows 2000	Unavailable
Windows XP and later	Kdexts.dll

Remarks

You can use **!whatperftime** to convert values retrieved by calling **QueryPerformanceCounter**. Performance counter time values are also found in software traces.

The output is displayed as *HH:MM:SS.mmm*. Here is an example:

```
kd> !whatperftime 304589
3163529 Performance Counter in Standard Time: .004.313s
```

!whea

12/5/2018 • 2 minutes to read • Edit Online

The **!whea** extension displays top-level Windows Hardware Error Architecture (WHEA) information.

```
!whea
```

DLL

Windows 2000	Unavailable
Windows XP	Unavailable
Windows Server 2003	Unavailable
Windows Vista and later	Kdexts.dll

This extension can be used only in Windows Vista and later versions of Windows.

Additional Information

The **!errrec** and **!errpkt** extensions can be used to display additional WHEA information. For general information about WHEA, see Windows Hardware Error Architecture (WHEA) in the Windows Driver Kit (WDK) documentation.

Remarks

The following example shows the (truncated) output of the **!whea** extension:

```
3: kd> !whea
Error Source Table @ fffff800019ca250
13 Error Sources
Error Source 0 @ fffffa80064132c0
   Notify Type      : Machine Check Exception
   Type             : 0x0 (MCE)
   Error Count      : 8
   Record Count     : 10
   Record Length    : c3e
   Error Records    : wrapper @ fffffa8007cf4000  record @ fffffa8007cf4030
                    : wrapper @ fffffa8007cf4c3e  record @ fffffa8007cf4c6e
                    : wrapper @ fffffa8007cf587c  record @ fffffa8007cf58ac
                    : wrapper @ fffffa8007cf64ba  record @ fffffa8007cf64ea
                    : wrapper @ fffffa8007cf70f8  record @ fffffa8007cf7128
                    : wrapper @ fffffa8007cf7d36  record @ fffffa8007cf7d66
                    : wrapper @ fffffa8007cf8974  record @ fffffa8007cf89a4
                    : wrapper @ fffffa8007cf95b2  record @ fffffa8007cf95e2
                    : wrapper @ fffffa8007cfa1f0  record @ fffffa8007cfa220
                    : wrapper @ fffffa8007cfae2e  record @ fffffa8007cfae5e
                    : wrapper @ fffffa8007cfba6c  record @ fffffa8007cfba9c
                    : wrapper @ fffffa8007cfc6aa  record @ fffffa8007cfc6da
                    : wrapper @ fffffa8007cfd2e8  record @ fffffa8007cfd318
                    : wrapper @ fffffa8007cfdf26  record @ fffffa8007cfdf56
```

```
                    : wrapper @ fffffa8007cfeb64  record @ fffffa8007cfeb94
                    : wrapper @ fffffa8007cff7a2  record @ fffffa8007cff7d2
   Descriptor       : @ fffffa8006413328
      Length                    : 3cc
      Max Raw Data Length       : 392
      Num Records To Preallocate : 10
      Max Sections Per Record   : 3
      Error Source ID           : 0
      Flags                     : 00000000
Error Source 1 @ fffffa8007d00bc0
   Notify Type      : Corrected Machine Check
   Type             : 0x1 (CMC)
   Error Count      : 0
   Record Count     : 10
   Record Length    : c3e
   Error Records    : wrapper @ fffffa8007d01000  record @ fffffa8007d01030
                    : wrapper @ fffffa8007d01c3e  record @ fffffa8007d01c6e
                    : wrapper @ fffffa8007d0287c  record @ fffffa8007d028ac
                    : wrapper @ fffffa8007d034ba  record @ fffffa8007d034ea
                    : wrapper @ fffffa8007d040f8  record @ fffffa8007d04128
                    : wrapper @ fffffa8007d04d36  record @ fffffa8007d04d66
                    : wrapper @ fffffa8007d05974  record @ fffffa8007d059a4
                    : wrapper @ fffffa8007d065b2  record @ fffffa8007d065e2
                    : wrapper @ fffffa8007d071f0  record @ fffffa8007d07220
                    : wrapper @ fffffa8007d07e2e  record @ fffffa8007d07e5e
                    : wrapper @ fffffa8007d08a6c  record @ fffffa8007d08a9c
                    : wrapper @ fffffa8007d096aa  record @ fffffa8007d096da
                    : wrapper @ fffffa8007d0a2e8  record @ fffffa8007d0a318
                    : wrapper @ fffffa8007d0af26  record @ fffffa8007d0af56
                    : wrapper @ fffffa8007d0bb64  record @ fffffa8007d0bb94
                    : wrapper @ fffffa8007d0c7a2  record @ fffffa8007d0c7d2
   Descriptor       : @ fffffa8007d00c28
      Length                    : 3cc
      Max Raw Data Length       : 392
      Num Records To Preallocate : 10
      Max Sections Per Record   : 3
      Error Source ID           : 1
      Flags                     : 00000000
Error Source 2 @ fffffa8007d00770
   Notify Type      : Non-Maskable Interrupt
   Type             : 0x3 (NMI)
   Error Count      : 1
   Record Count     : 1
   Record Length    : 1f8
   Error Records    : wrapper @ fffffa800631b2c0  record @ fffffa800631b2f0
   Descriptor       : @ fffffa8007d007d8
      Length                    : 3cc
      Max Raw Data Length       : 100
      Num Records To Preallocate : 1
      Max Sections Per Record   : 1
      Error Source ID           : 2
      Flags                     : 00000000
Error Source 3 @ fffffa8007d0dbc0
   Notify Type      : BOOT Error Record
   Type             : 0x7 (BOOT)
   Error Count      : 0
   Record Count     : 1
   Record Length    : 4f8
   Error Records    : wrapper @ 0000000000000000  record @ 0000000000000030
   Descriptor       : @ fffffa8007d0dc28
      Length                    : 3cc
      Max Raw Data Length       : 400
      Num Records To Preallocate : 1
      Max Sections Per Record   : 1
      Error Source ID           : 3
      Flags                     : 00000000

. . . .
```

!wsle

8/26/2019 • 2 minutes to read • Edit Online

The **!wsle** extension displays all working set list entries (WSLEs).

Syntax

```
!wsle [Flags [Address]]
```dbgcmd

## <span id="ddk__wsle_dbg"></span><span id="DDK__WSLE_DBG"></span>Parameters


<span id="_______Flags______"></span><span id="_______flags______"></span><span id="_______FLAGS______"></span>
*Flags*
Specifies the information to include in the display. This can be any combination of the following bits. The default is zero. If this is used, only basic information about the working set is displayed.

<span id="Bit_0__0x1_"></span><span id="bit_0__0x1_"></span><span id="BIT_0__0X1_"></span>Bit 0 (0x1)
Causes the display to include information about each WSLE's address, age, lock status, and reference count. If a WSLE has an invalid page table entry (PTE) or page directory entry (PDE) associated with it, this is also displayed.

<span id="Bit_1__0x2_"></span><span id="bit_1__0x2_"></span><span id="BIT_1__0X2_"></span>Bit 1 (0x2)
Causes the display to include the total number of valid WSLEs, the index of the last WSLE, and the index of the first free WSLE.

<span id="Bit_2__0x4_"></span><span id="bit_2__0x4_"></span><span id="BIT_2__0X4_"></span>Bit 2 (0x4)
Causes the display to include the total number of free WSLEs, as well as the index of each free WSLE. If bit 1 is also set, then a check is done to make sure that the number of free WSLEs plus the number of valid WSLEs is actually equal to the total number of WSLEs.

<span id="_______Address______"></span><span id="_______address______"></span><span id="_______ADDRESS______">
</span> *Address*
Specifies the address of the working set list. If this is omitted, the default working set list is used. Specifying zero for *Address* is the same as omitting it.

### <span id="DLL"></span><span id="dll"></span>DLL

Kdexts.dll

### <span id="Additional_Information"></span><span id="additional_information"></span><span id="ADDITIONAL_INFORMATION"></span>Additional Information

For information about working sets, see the Windows Driver Kit (WDK) documentation and *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks
-------

This extension can take a significant amount of time to execute.

Here is an example from an x86 target computer running Windows Server 2003:

```dbgcmd
kd> !wsle 3

Working Set @ c0503000
    FirstFree:       a7  FirstDynamic:          4
    LastEntry      23d  NextSlot:         4  LastInitialized      259
    NonDirect       65  HashTable:        0  HashTableSize:         0
```

```
Reading the WSLE data...

Virtual Address           Age  Locked  ReferenceCount
        c0300203          0        1        1
        c0301203          0        1        1
        c0502203          0        1        1
        c0503203          0        1        1
        c01ff201          0        0        1
        77f74d19          3        0        1
        7ffdfa01          2        0        1
        c0001201          0        0        1

.....

Reading the WSLE data...
Valid WSLE entries = 0xa7
found end @ wsle index 0x259

.....
```

!zombies

4/8/2019 • 2 minutes to read • Edit Online

The **!zombies** extension displays all dead ("zombie") processes or threads.

```
!zombies [Flags [RestartAddress]]
```

Parameters

Flags
Specifies what will be displayed. Possible values include:

1
Displays all zombie processes. (This is the default.)

2
Displays all zombie threads.

RestartAddress
Specifies the hexadecimal address at which to begin the search. This is useful if the previous search was terminated prematurely. The default is zero.

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Unavailable

Additional Information

To see a list of all processes and threads, use the **!process** extension.

For general information about processes and threads in kernel mode, see Changing Contexts. For more information about analyzing processes and threads, see *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

Zombie processes are dead processes that have not yet been removed from the process list. Zombie threads are analogous.

This extension is available only for Windows 2000.

User-Mode Extensions

12/5/2018 • 2 minutes to read • Edit Online

This section of the reference describes extension commands that are primarily used during user-mode debugging.

The debugger will automatically load the proper version of these extension commands. Unless you have manually loaded a different version, you do not have to keep track of the DLL versions being used. See Using Debugger Extension Commands for a description of the default module search order. See Loading Debugger Extension DLLs for an explanation of how to load extension modules.

Each extension command reference page lists the DLLs that expose that command. Use the following rules to determine the proper directory from which to load this extension DLL:

- If your target application is running on Windows XP or a later version of Windows, use winxp\Ntsdexts.dll.

In addition, user-mode extensions that are not specific to any single operating system can be found in winext\Uext.dll.

!avrf

6/7/2019 • 5 minutes to read • Edit Online

The **!avrf** extension controls the settings of Application Verifier and displays a variety of output produced by Application Verifier.

```
!avrf
!avrf -vs { Length | -a Address }
!avrf -hp { Length | -a Address }
!avrf -cs { Length | -a Address }
!avrf -dlls [ Length ]
!avrf -trm
!avrf -ex [ Length ]
!avrf -threads [ ThreadID ]
!avrf -tp [ ThreadID ]
!avrf -srw  [ Address | Address Length ] [ -stats ]
!avrf -leak  [ -m ModuleName] [ -r ResourceType] [ -a Address ] [ -t ]
!avrf -trace TraceIndex
!avrf -cnt
!avrf -brk [BreakEventType]
!avrf -flt [EventType Probability]
!avrf -flt break EventType
!avrf -flt stacks Length
!avrf -trg [ Start End | dll Module | all ]
!avrf -settings
!avrf -skp [ Start End | dll Module | all | Time ]
```

Parameters

-vs { *Length* **| -a** *Address* **}**
Displays the virtual space operation log. *Length* specifies the number of records to display, starting with the most recent. *Address* specifies the virtual address. Records of the virtual operations that contain this virtual address are displayed.

-hp { *Length* **| -a** *Address* **}**
Displays the heap operation log. *Address* specifies the heap address. Records of the heap operations that contain this heap address are displayed.

-cs { *Length* **| -a** *Address* **}**
Displays the critical section delete log. *Length* specifies the number of records to display, starting with the most recent. *Address* specifies the critical section address. Records for the particular critical section are displayed when *Address* is specified.

-dlls [*Length* **]**
Displays the DLL load/unload log. *Length* specifies the number of records to display, starting with the most recent.

-trm
Displays a log of all terminated and suspended threads.

-ex [*Length* **]**
Displays the exception log. Application Verifier tracks all the exceptions in the application.

-threads [*ThreadID* **]**
Displays information about threads in the target process. For child threads, the stack size and the **CreateThread** flags specified by the parent are also displayed. If you provide a thread ID, information for only that thread is

displayed.

-tp [*ThreadID* **]**
Displays the threadpool log. This log contains stack traces for various operations such as changing the thread affinity mask, changing thread priority, posting thread messages, and initializing or uninitializing COM from within the threadpool callback. If you provide a thread ID, information for that thread only is displayed.

-srw [*Address* **|** *Address Length* **] [-stats]**
Displays the Slim Reader/Writer (SRW) log. If you specify *Address*, records for the SRW lock at that address are displayed. If you specify *Address* and *Length*, records for SRW locks in that address range are displayed. If you include the **-stats** option, the SRW lock statistics are displayed.

-leak [-m *ModuleName***] [-r** *ResourceType***] [-a** *Address* **] [-t]**
Displays the outstanding resources log. These resources may or may not be leaks at any given point. If you specify *Modulename* (including the extension), all outstanding resources in the specified module are displayed. If you specify *ResourceType*, all outstanding resources of that resource type are displayed. If you specify *Address*, records of outstanding resources with that address are displayed. *ResourceType* can be one of the following:

Heap: Displays heap allocations using Win32 Heap APIs

Local: Displays Local/Global allocations

CRT: Displays allocations using CRT APIs

Virtual: Displays Virtual reservations

BSTR: Displays BSTR allocations

Registry: Displays Registry key opens

Power: Displays power notification objects

Handle: Displays thread, file, and event handle allocations

-trace *TraceIndex* Displays a stack trace for the specified trace index. Some structures use this 16-bit index number to identify a stack trace. This index points to a location within the stack trace database.

-cnt Displays a list of global counters.

-brk [*BreakEventType* **]** Specifies a break event. *BreakEventType* is the type number of the break event. For a list of possible types, and a list of the current break event settings, enter **!avrf -brk**.

-flt [*EventType Probability* **]** Specifies a fault injection. *EventType* is the type number of the event. *Probability* is the frequency with which the event will fail. This can be any integer between 0 and 1,000,000 (0xF4240). If you enter **!avrf -flt** with no additional parameters, the current fault injection settings are displayed.

-**flt break** *EventType* Causes Application Verifier to break into the debugger each time this fault, specified by *EventType*, is injected.

-flt stacks *Length* Displays *Length* number of stack traces for the most recent fault-injected operations.

-trg [*Start End* **| dll** *Module* **| all]** Specifies a target range. *Start* is the beginning address of the target range. *End* is the ending address of the target range. *Module* specifies the name (including the .exe or .dll extension, but not including the path) of a module to be targeted. If you enter **-trg all**, all target ranges are reset. If you enter **-trg** with no additional parameters, the current target ranges are displayed.

-skp [*Start End* **| dll** *Module* **| all |** *Time* **]** Specifies an exclusion range. *Start* is the beginning address of the exclusion range. *End* is the ending address of the exclusion range. Module specifies the name of a module to be targeted or excluded. *Module* specifies the name (including the .exe or .dll extension, but not including the path) of a module to be excluded. If you enter **-skp all**, all target ranges or exclusion ranges are reset. If you enter a*Time*

value, all faults are suppressed for *Time* milliseconds after execution resumes.

DLL

exts.dll

Additional Information

For information about how to download and install Application Verifier and its documentation, see Application Verifier.

Remarks

When the **!avrf** extension is used with no parameters, it displays the current Application Verifier options. If the **Full page heap** or **Fast fill heap** option has been enabled, information about active page heaps is displayed as well. For some examples, see "Heap Operation Logs" in the Application Verifier documentation.

If an Application Verifier Stop has occurred, the **!avrf** extension with no parameters will reveal the nature of the stop and its cause. For some examples, see "Debugging Application Verifier Stops" in the Application Verifier documentation.

If symbols for ntdll.dll and verifier.dll are missing, the **!avrf** extension generates an error message. For information about how to address this problem, see "Setting Up a Debugger for Application Verifier" in the Application Verifier documentation.

!critsec

12/5/2018 • 2 minutes to read • Edit Online

The **!critsec** extension displays a critical section.

```
!critsec Address
```

Parameters

Address
Specifies the hexadecimal address of the critical section.

DLL

Windows 2000	Ntsdexts.dll
Windows XP and later	Ntsdexts.dll

Additional Information

For other commands and extensions that can display critical section information, see Displaying a Critical Section. For information about critical sections, see the Microsoft Windows SDK documentation, the Windows Driver Kit (WDK) documentation, and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

If you do not know the address of the critical section, you should use the **!ntsdexts.locks** extension. This displays all critical sections that have been initialized by calling **RtlInitializeCriticalSection**.

Here is an example:

```
0:000> !critsec 3a8c0e9c

CritSec +3a8c0e9c at 3A8C0E9C
LockCount          1
RecursionCount     1
OwningThread       99
EntryCount         472
ContentionCount    1
*** Locked
```

!dp (!ntsdexts.dp)

12/5/2018 • 2 minutes to read • Edit Online

The **!dp** extension in Ntsdexts.dll displays a CSR process.

This extension command should not be confused with the **dp (Display Memory)** command, or with the **!kdext*.dp** extension command.

```
!dp [v] [ PID | CSR-Process ]
```

Parameters

v
Verbose mode. Causes the display to include structure and thread list.

PID
Specifies the process ID of the CSR process.

CSR-Process
Specifies the hexadecimal address of the CSR process.

DLL

Windows 2000	Ntsdexts.dll
Windows XP and later	Ntsdexts.dll

Remarks

This extension displays the process address, process ID, sequence number, flags, and reference count. If verbose mode is selected, additional details are displayed, and thread information is shown for each process.

If no process is specified, all processes are displayed.

See also

!dt

!dreg

8/26/2019 • 2 minutes to read • Edit Online

The **!dreg** extension displays registry information.

```
!dreg [-d|-w] KeyPath[!Value]
!dreg
```

Parameters

-d
Causes binary values to be displayed as DWORDs.

-w
Causes binary values to be displayed as WORDs.

KeyPath
Specifies the registry key path. It can begin with any of the following abbreviations:

hklm
HKEY_LOCAL_MACHINE

hkcu
HKEY_CURRENT_USER

hkcr
HKEY_CLASSES_ROOT

hku
HKEY_USERS

If no abbreviation is used, HKEY_LOCAL_MACHINE is assumed.

Value
Specifies the name of the registry value to be displayed. If an asterisk (*) is used, all values are displayed. If *Value* is omitted, all subkeys are displayed.

DLL

Windows 2000	Ntsdexts.dll
Windows XP and later	Ntsdexts.dll

Additional Information

For information about the registry, see the Windows Driver Kit (WDK) documentation and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

The **!dreg** extension can be used to display the registry during user-mode debugging.

It is most useful during remote debugging, as it allows you to browse the registry of the remote machine. It is also useful when controlling the user-mode debugger from the kernel debugger, because you cannot run a standard registry editor on the target machine when it is frozen. (You can use the **.sleep** command for this purpose as well. See Controlling the User-Mode Debugger from the Kernel Debugger for details.)

It is also useful when debugging locally, as the information is presented in an easily readable format.

If **!dreg** is used during kernel-mode debugging, the results shown will be for the host computer, and not the target computer. To display raw registry information for the target computer, use the **!reg** extension instead.

Here are some examples. The following will display all subkeys of the specified registry key:

```
!dreg hkcu\Software\Microsoft
```

The following will display all values in the specified registry key:

```
!dreg System\CurrentControlSet\Services\Tcpip!*
```

The following will display the value Start in the specified registry key:

```
!dreg System\CurrentControlSet\Services\Tcpip!Start
```

Typing **!dreg** without any arguments will display some brief Help text for this extension in the Debugger Command window.

!dt

12/5/2018 • 2 minutes to read • Edit Online

The **!dt** extension displays information about a CSR thread.

This extension command should not be confused with the **dt (Display Type)** command.

```
!dt [v] CSR-Thread
```

Parameters

v
Verbose mode.

CSR-Thread
Specifies the hexadecimal address of the CSR thread.

DLL

Windows 2000	Ntsdexts.dll
Windows XP and later	Ntsdexts.dll

Remarks

This extension displays the thread, process, client ID, flags, and reference count associated with the CSR thread. If verbose mode is selected, the display will also include list pointers, thread handle, and the wait block.

See also

!dp (!ntsdexts.dp)

!findstack

12/5/2018 • 2 minutes to read • Edit Online

The **!findstack** extension locates all of the stacks that contain a specified symbol or module.

```
!findstack Symbol [DisplayLevel]
!findstack -?
```

Parameters

Symbol
Specifies a symbol or module.

DisplayLevel
Specifies what the display should contain. This can be any one of the following values. The default value is 1.

0
Displays only the thread ID for each thread that contains *Symbol*.

1
Displays both the thread ID and the frame for each thread that contains *Symbol*.

2
Displays the entire thread stack for each thread that contains *Symbol*.

-?
Displays some brief Help text for this extension in the Debugger Command window.

DLL

Windows 2000	Uext.dll
Windows XP and later	Uext.dll

Additional Information

For more information about stack traces, see the **k, kb, kc, kd, kp, kP, kv (Display Stack Backtrace)** commands.

Remarks

The **!stacks** kernel-mode extension also display information about stacks, including a brief summary of the state of every thread.

The following are some examples of the output from this extension:

```
0:023> !uext.findstack wininet
Thread 009, 2 frame(s) match
        * 06 03eaffac 771d9263 wininet!ICAsyncThread::SelectThread+0x22a
        * 07 03eaffb4 7c80b50b wininet!ICAsyncThread::SelectThreadWrapper+0xd

Thread 011, 2 frame(s) match
        * 04 03f6ffb0 771cda1d wininet!AUTO_PROXY_DLLS::DoThreadProcessing+0xa1
        * 05 03f6ffb4 7c80b50b wininet!AutoProxyThreadFunc+0xb

Thread 020, 6 frame(s) match
        * 18 090dfde8 771db73a wininet!CheckForNoNetOverride+0x9c
        * 19 090dfe18 771c5e4d wininet!InternetAutodialIfNotLocalHost+0x220
        * 20 090dfe8c 771c5d6a wininet!ParseUrlForHttp_Fsm+0x135
        * 21 090dfe98 771bcb2c wininet!CFsm_ParseUrlForHttp::RunSM+0x2b
        * 22 090dfeb0 771d734a wininet!CFsm::Run+0x39
        * 23 090dfee0 77f6ad84 wininet!CFsm::RunWorkItem+0x79

Thread 023, 9 frame(s) match
        * 16 0bd4fe00 771bd256 wininet!ICSocket::Connect_Start+0x17e
        * 17 0bd4fe0c 771bcb2c wininet!CFsm_SocketConnect::RunSM+0x42
        * 18 0bd4fe24 771bcada wininet!CFsm::Run+0x39
        * 19 0bd4fe3c 771bd22b wininet!DoFsm+0x25
        * 20 0bd4fe4c 771bd706 wininet!ICSocket::Connect+0x32
        * 21 0bd4fe8c 771bd4cb wininet!HTTP_REQUEST_HANDLE_OBJECT::OpenConnection_Fsm+0x391
        * 22 0bd4fe98 771bcb2c wininet!CFsm_OpenConnection::RunSM+0x33
        * 23 0bd4feb0 771d734a wininet!CFsm::Run+0x39
        * 24 0bd4fee0 77f6ad84 wininet!CFsm::RunWorkItem+0x79

0:023> !uext.findstack wininet!CFsm::Run 0
Thread 020, 2 frame(s) match
Thread 023, 3 frame(s) match

0:023> !uext.findstack wininet!CFsm 0
Thread 020, 3 frame(s) match
Thread 023, 5 frame(s) match
```

!gatom

12/5/2018 • 2 minutes to read • Edit Online

The **!gatom** extension displays the global atom table.

```
!gatom
```

DLL

Windows 2000	Ntsdexts.dll
Windows XP and later	Ntsdexts.dll

Additional Information

For information about the global atom table, see the Microsoft Windows SDK documentation.

!igrep

12/5/2018 • 2 minutes to read • Edit Online

The **!igrep** extension searches for a pattern in disassembly.

```
!igrep [Pattern [StartAddress]]
```

Parameters

Pattern
Specifies the pattern to search for. If omitted, the last *Pattern* is used.

StartAddress
Specifies the hexadecimal address at which to begin searching. If omitted, the current program counter is used.

DLL

Windows 2000	Ntsdexts.dll
Windows XP and later	Unavailable

!locks (!ntsdexts.locks)

1/3/2019 • 2 minutes to read • Edit Online

The **!locks** extension in Ntsdexts.dll displays a list of critical sections associated with the current process.

This extension command should not be confused with the **!kdext*.locks** extension command.

```
!locks [Options]
```

Parameters

Options
Specifies the amount of information to be displayed. Any combination of the following options can be used:

-v
Causes the display to include all critical sections, even those that are not currently owned.

-o
Causes the display to only include orphaned information (pointers that do not actually point to valid critical sections).

DLL

Windows 2000	Ntsdexts.dll
Windows XP and later	Ntsdexts.dll

Additional Information

For other commands and extensions that can display critical section information, see Displaying a Critical Section. For information about critical sections, see the Microsoft Windows SDK documentation, the Windows Driver Kit (WDK) documentation, and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

This extension command shows all critical sections that have been initialized by calling **RtlInitializeCriticalSection**. If there are no critical sections, then no output will result.

Here is an example:

```
0:000> !locks

CritSec w3svc!g_pWamDictator+a0 at 68C2C298
LockCount          0
RecursionCount     1
OwningThread       d1
EntryCount         1
ContentionCount    0
*** Locked

CritSec SMTPSVC+66a30 at 67906A30
LockCount          0
RecursionCount     1
OwningThread       d0
EntryCount         1
ContentionCount    0
*** Locked
```

!mapped_file

12/5/2018 • 2 minutes to read • Edit Online

The **!mapped_file** extension displays the name of the file that backs the file mapping that contains a specified address.

```
!mapped_file Address
```

Parameters

Address
Specifies the address of the file mapping. If *Address* is not in a mapping, the command fails.

DLL

Windows 2000	Uext.dll
Windows XP and later	Uext.dll

The **!mapped_file** extension can only be used during live, nonremote debugging.

Additional Information

For more information about file mapping, see MapViewOfFile in the Windows SDK.

Remarks

Here are three examples. The first two addresses used are mapped from a file, and the third is not.

```
0:000> !mapped_file 4121ec
Mapped file name for 004121ec: '\Device\HarddiskVolume2\CODE\TimeTest\Debug\TimeTest.exe'

0:000> !mapped_file 77150000
Mapped file name for 77150000: '\Device\HarddiskVolume2\Windows\System32\kernel32.dll'

0:000> !mapped_file 80310000
No information found for 80310000: error 87
```

!runaway

4/8/2019 • 2 minutes to read • Edit Online

The **!runaway** extension displays information about the time consumed by each thread.

```
!runaway [Flags]
```

Parameters

Flags
Specifies the kind of information to be displayed. *Flags* can be any combination of the following bits. The default value is 0x1.

Bit 0 (0x1)
Causes the debugger to show the amount of user time consumed by each thread.

Bit 1 (0x2)
Causes the debugger to show the amount of kernel time consumed by each thread.

Bit 2 (0x4)
Causes the debugger to show the amount of time that has elapsed since each thread was created.

DLL

Windows 2000	Uext.dll Ntsdexts.dll
Windows XP and later	Uext.dll Ntsdexts.dll

The **!runaway** extension can only be used during live debugging or when debugging crash dump files created by **.dump /mt** or **.dump /ma**.

Additional Information

For information about threads in user mode, see Controlling Processes and Threads. For more information about analyzing processes and threads, see *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

This extension is a quick way to find out which threads are spinning out of control or consuming too much CPU time.

The display identifies each thread by the debugger's internal thread numbering and by the thread ID in hexadecimal. The debugger IDs are also shown.

Here is an example:

```
0:001> !runaway 7

 User Mode Time
 Thread       Time
 0:55c        0:00:00.0093
 1:1a4        0:00:00.0000

 Kernel Mode Time
 Thread       Time
 0:55c        0:00:00.0140
 1:1a4        0:00:00.0000

 Elapsed Time
 Thread       Time
 0:55c        0:00:43.0533
 1:1a4        0:00:25.0876
```

!threadtoken

8/26/2019 • 2 minutes to read • Edit Online

The **!threadtoken** extension displays the impersonation state of the current thread.

```
!threadtoken
```

DLL

Windows 2000	Ntsdexts.dll
Windows XP and later	Unavailable

Additional Information

For information about threads and impersonation, see the Microsoft Windows SDK documentation and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

The **!threadtoken** extension is obsolete in Windows XP and later versions of Windows. Use **!token** instead.

If the current thread is impersonating, the token that this thread is using will be displayed.

Otherwise, a message reading "Thread is not impersonating" will appear. The process token will then be displayed.

Tokens will be displayed in the same format that **!handle** uses when displaying token handles.

Here is an example:

```
0:000> ~
.  0  id: 1d0.55c   Suspend: 1 Teb 7ffde000 Unfrozen
#  1  id: 1d0.1a4   Suspend: 1 Teb 7ffdd000 Unfrozen

0:000> !threadtoken

***Thread is not impersonating, using process token***
    Auth Id    0 : 0x1c93d
    Type       Primary
    Imp Level  Anonymous
     Token Id  0 : 0x5e8c19
     Mod Id    0 : 0x5e8c12
     Dyn Chg   0x1f4
     Dyn Avail 0x1a4
     Groups    26
     Privs     17
     User      S-1-5-21-2127521184-1604012920-1887927527-74790
     Groups    26
               S-1-5-21-2127521184-1604012920-1887927527-513
               S-1-1-0
               S-1-5-32-544
               S-1-5-32-545
               S-1-5-21-2127521184-1604012920-1887927527-277551
               S-1-5-21-2127521184-1604012920-1887927527-211604
               S-1-5-21-2127521184-1604012920-1887927527-10546
               S-1-5-21-2127521184-1604012920-1887927527-246657
               S-1-5-21-2127521184-1604012920-1887927527-277552
               S-1-5-21-2127521184-1604012920-1887927527-416040
               S-1-5-21-2127521184-1604012920-1887927527-96548
               S-1-5-21-2127521184-1604012920-1887927527-262644
               S-1-5-21-2127521184-1604012920-1887927527-155802
               S-1-5-21-2127521184-1604012920-1887927527-158763
               S-1-5-21-2127521184-1604012920-1887927527-279132
               S-1-5-21-2127521184-1604012920-1887927527-443952
               S-1-5-21-2127521184-1604012920-1887927527-175772
               S-1-5-21-2127521184-1604012920-1887927527-388472
               S-1-5-21-2127521184-1604012920-1887927527-443950
               S-1-5-21-2127521184-1604012920-1887927527-266975
               S-1-5-21-2127521184-1604012920-1887927527-158181
               S-1-5-21-2127521184-1604012920-1887927527-279435
               S-1-5-5-0-116804
               S-1-2-0
               S-1-5-4
               S-1-5-11
     Privileges    17
               SeUndockPrivilege ( Enabled Default )
               SeTakeOwnershipPrivilege ( )
               SeShutdownPrivilege ( )
               SeDebugPrivilege ( )
               SeIncreaseBasePriorityPrivilege ( )
               SeAuditPrivilege ( )
               SeSyncAgentPrivilege ( )
               SeLoadDriverPrivilege ( )
               SeSystemEnvironmentPrivilege ( Enabled )
               SeRemoteShutdownPrivilege ( )
               SeProfileSingleProcessPrivilege ( )
               SeCreatePagefilePrivilege ( )
               SeCreatePermanentPrivilege ( )
               SeSystemProfilePrivilege ( Enabled )
               SeBackupPrivilege ( )
               SeMachineAccountPrivilege ( )
               SeEnableDelegationPrivilege ( Enabled )
```

!uniqstack

12/5/2018 • 2 minutes to read • Edit Online

The **!uniqstack** extension displays all of the stacks for all of the threads in the current process, excluding stacks that appear to have duplicates.

```
!uniqstack [ -b | -v | -p ] [ -n ]
```

Parameters

-b
Causes the display to include the first three parameters passed to each function.

-v
Causes the display to include frame pointer omission (FPO) information. On x86-based processors, the calling convention information is also displayed.

-p
Causes the display to include the full parameters for each function that is called in the stack trace. This listing will include each parameter's data type, name, and value. *This requires full symbol information.*

-n
Causes frame numbers to be displayed.

DLL

Windows 2000	Uext.dll
Windows XP and later	Uext.dll

Remarks

This extension is similar to the **k, kb, kc, kd, kp, kP, kv (Display Stack Backtrace)** command, except that it does not display duplicate stacks.

For example:

```
0:000> !uniqstack
Processing 14 threads, please wait

.  0  Id: f0f0f0f0.15c Suspend: 1 Teb: 00000000`7fff8000 Unfrozen
      Priority: 0
Child-SP          Child-BSP         RetAddr
00000000`0006e5e0 00000000`00070420 00000000`6b009840
00000000`0006e600 00000000`000703d0 00000000`6b03db00
00000000`0006e950 00000000`000703b0 00000000`6b008520
00000000`0006e950 00000000`00070368 00000000`6b1845e0
00000000`0006e9b0 00000000`00070310 00000000`6b009980
00000000`0006e9d0 00000000`000702d0 00000000`6b009ff0
00000000`0006e9e0 00000000`00070248 00000000`77ea4a10
00000000`0006f290 00000000`000700e0 00000000`77ea5d60
00000000`0006f4c0 00000000`00070028 00000000`77ed6000
00000000`0006f550 00000000`00070000 00000000`77ed9000
00000000`0006f550 00000000`00070000 00000000`77ca78a0
00000000`0006fff0 00000000`00070000 00000000`00000000

.  1  Id: f0f0f0f0.718 Suspend: 1 Teb: 00000000`7fff4000 Unfrozen
      Priority: 0
Child-SP          Child-BSP         RetAddr
00000000`0043eb50 00000000`00440250 00000000`6b008520
00000000`0043eb80 00000000`00440208 00000000`6b1845e0
00000000`0043ebe0 00000000`004401a8 00000000`6b009980
00000000`0043ec00 00000000`00440168 00000000`6b009ff0
00000000`0043ec10 00000000`004400e0 00000000`77ea5f50
00000000`0043f4c0 00000000`00440028 00000000`77ed6000
00000000`0043f550 00000000`00440000 00000000`77ed9000
00000000`0043f550 00000000`00440000 00000000`7de05690
00000000`0043fff0 00000000`00440000 00000000`00000000

. 13  Id: f0f0f0f0.494 Suspend: 1 Teb: 00000000`7ef98000 Unfrozen
      Priority: 0
Child-SP          Child-BSP         RetAddr
00000000`00feffe0 00000000`00ff0040 00000000`77e94f30
00000000`00feffe0 00000000`00ff0040 00000000`00000000

Total threads: 14
Duplicate callstacks: 11 (windbg thread #s follow):
2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
```

!vadump

12/5/2018 • 2 minutes to read • Edit Online

The **!vadump** extension displays all virtual memory ranges and their corresponding protection information.

```
!vadump [-v]
```

Parameters

-v

Causes the display to include information about each original allocation region as well. Because individual addresses within a region can have their protection altered after memory is allocated (by **VirtualProtect**, for example), the original protection status for this larger region may not be the same as that of each range within the region.

DLL

Windows 2000	Uext.dll
Windows XP and later	Uext.dll

Additional Information

To view memory protection information for a single virtual address, use **!vprot**. For information about memory protection, see *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

Here is an example:

```
0:000> !vadump
BaseAddress:       00000000
RegionSize:        00010000
State:             00010000  MEM_FREE
Protect:           00000001  PAGE_NOACCESS

BaseAddress:       00010000
RegionSize:        00001000
State:             00001000  MEM_COMMIT
Protect:           00000004  PAGE_READWRITE
Type:              00020000  MEM_PRIVATE
.........
```

In this display, the State line shows the state of the memory range beginning at the specified BaseAddress. The possible state values are MEM_COMMIT, MEM_FREE, and MEM_RESERVE.

The Protect line shows the protection status of this memory range. The possible protection values are PAGE_NOACCESS, PAGE_READONLY, PAGE_READWRITE, PAGE_EXECUTE, PAGE_EXECUTE_READ, PAGE_EXECUTE_READWRITE, PAGE_WRITECOPY, PAGE_EXECUTE_WRITECOPY, and PAGE_GUARD.

The Type line shows the memory type. The possible values are MEM_IMAGE, MEM_MAPPED, and

MEM_PRIVATE.

Here is an example using the **-v** parameter:

```
0:000> !vadump -v
BaseAddress:       00000000
AllocationBase:    00000000
RegionSize:        00010000
State:             00010000  MEM_FREE
Protect:           00000001  PAGE_NOACCESS

BaseAddress:       00010000
AllocationBase:    00010000
AllocationProtect: 00000004  PAGE_READWRITE
RegionSize:        00001000
State:             00001000  MEM_COMMIT
Protect:           00000004  PAGE_READWRITE
Type:              00020000  MEM_PRIVATE
.........
```

When **-v** is used, the AllocationProtect line shows the default protection that the entire region was created with. The Protect line shows the actual protection for this specific address.

!vprot

4/8/2019 • 2 minutes to read • Edit Online

The **!vprot** extension displays virtual memory protection information.

```
!vprot [Address]
```

Parameters

Address
Specifies the hexadecimal address whose memory protection status is to be displayed.

DLL

Windows 2000	Uext.dll Ntsdexts.dll
Windows XP and later	Uext.dll

Additional Information

To view memory protection information for all memory ranges owned by the target process, use **!vadump**. For information about memory protection, see *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

The **!vprot** extension command can be used for both live debugging and dump file debugging.

Here is an example:

```
0:000> !vprot 30c191c
BaseAddress: 030c1000
AllocationBase: 030c0000
AllocationProtect: 00000080 PAGE_EXECUTE_WRITECOPY
RegionSize: 00011000
State: 00001000 MEM_COMMIT
Protect: 00000010 PAGE_EXECUTE
Type: 01000000 MEM_IMAGE
```

In this display, the AllocationProtect line shows the default protection that the entire region was created with. Note that individual addresses within this region can have their protection altered after memory is allocated (for example, if **VirtualProtect** is called). The Protect line shows the actual protection for this specific address. The possible protection values are PAGE_NOACCESS, PAGE_READONLY, PAGE_READWRITE, PAGE_EXECUTE, PAGE_EXECUTE_READ, PAGE_EXECUTE_READWRITE, PAGE_WRITECOPY, PAGE_EXECUTE_WRITECOPY, and PAGE_GUARD.

The State line also applies to the specific virtual address passed to **!vprot**. The possible state values are MEM_COMMIT, MEM_FREE, and MEM_RESERVE.

The Type line shows the memory type. The possible values are MEM_IMAGE, MEM_MAPPED, and MEM_PRIVATE.

Specialized Extensions

12/5/2018 • 2 minutes to read • Edit Online

This section of the reference discusses extension commands in the extension DLLs that are used less often.

In this section

- Storage Kernel Debugger Extensions
- Bluetooth Extensions (Bthkd.dll)
- GPIO Extensions
- USB 3.0 Extensions
- USB 2.0 Debugger Extensions
- RCDRKD Extensions
- HID Extensions
- Logger Extensions (Logexts.dll)
- NDIS Extensions (Ndiskd.dll)
- RPC Extensions (Rpcexts.dll)
- ACPI Extensions (Acpikd.dll and Kdexts.dll)
- Graphics Driver Extensions (Gdikdx.dll)
- Kernel Streaming Extensions (Ks.dll)
- SCSI Miniport Extensions (Scsikd.dll and Minipkd.dll)
- Windows Driver Framework Extensions (Wdfkd.dll)
- User-Mode Driver Framework Extensions (Wudfext.dll)
- WMI Tracing Extensions (Wmitrace.dll)
- OEM Support Extensions (kdex2x86.dll)
- SieExtPub.dll

Storage Kernel Debugger Extensions

12/5/2018 • 2 minutes to read • Edit Online

The storage kernel debugger extensions (storagekd) are used for debugging the storage drivers on Windows 8 and above operating system (OS) targets.

Extension commands that are useful for debugging storage drivers, via classpnp managed storage class drivers and Storport managed storage miniport drivers, can be found in **Storagekd.dll**.

Please refer to SCSI Miniport Extensions (Scsikd.dll and Minipkd.dll) for debugging needs for Windows 7 and below version of OS targets.

Important You need special symbols to use this extension. For more information, see Debugging Tools for Windows.

Storage kernel debugger extension commands

COMMAND	DESCRIPTION
!storagekd.storhelp	Displays help text for **Storagekd.dll** extension commands.
!storagekd.storclass	Displays information about the specified *classpnp* device.
!storagekd.storadapter	Displays information about the specified *Storport* adapter.
!storagekd.storunit	Displays information about the specified *Storport* logical unit.
!storagekd.storloglist	Displays the *Storport* adapter's internal log entries.
!storagekd.storlogirp	Displays the *Storport's* internal log entries for the adapter filtered for the IRP provided.
!storagekd.storlogsrb	Displays the *Storport's* internal log entries for the adapter filtered for the Storage (or SCSI) Request Block (SRB) provided.
!storagekd.storsrb	Displays information about the specified Storage (or SCSI) Request Block (SRB).

!storagekd.storadapter

12/5/2018 • 2 minutes to read • Edit Online

The **!storagekd.storadapter** extension displays information about the specified Storport adapter.

```
!storagekd.storadapter [Address]
```

Parameters

Address
Specifies the address of a Storport adapter device object. If *Address* is omitted, a list of all Storport adapters is displayed.

DLL

Windows 8 and later	Storagekd.dll

Remarks

Here is an example of the **!storagekd.storadapter** display:

1: kd> ! storagekd.storadapter

```
# STORPORT adapters:
==================
## Driver                 Object            Extension          State
-----------------------------------------------------------------
\Driver\vhdmp          fffffa800649a050  fffffa800649a1a0   Working
```

1: kd> ! storagekd.storadapter fffffa800649a050

```
ADAPTER
   DeviceObj : fffffa800649a050   AdapterExt: fffffa800649a1a0   DriverObj :  fffffa800507fcb0
DeviceState : Working
   LowerDO fffffa8005f71e10   PhysicalDO fffffa8005f71e10
   SlowLock Free   RemLock -666
   SystemPowerState: Working AdapterPowerState D0   Full Duplex
   Bus 0   Slot 0   DMA 0000000000000000   Interrupt 0000000000000000
   Allocated ResourceList 0000000000000000
Translated ResourceList 0000000000000000
   Gateway: Outstanding 0   Lower 256   High 256
   PortConfigInfo fffffa800649a2d0
   HwInit fffffa80062e8840   HwDeviceExt fffffa8004b84d70   (112 bytes)
   SrbExt 2256 bytes   LUExt 24 bytes

Normal Logical Units:
   Product                 SCSI ID  Object            Extension          Pnd Out Ct State
   ------------------------------------------------------------------------------------------
   Msft       Virtual Di   0  0  1  fffffa800658a060  fffffa800658a1b0    0   0  0  Working

   Zombie Logical Units:
   Product                 SCSI ID  Object            Extension          Pnd Out Ct State
   ------------------------------------------------------------------------------------------
```

!storagekd.storclass

12/5/2018 • 2 minutes to read • Edit Online

The **!storagekd.storclass** extension displays information about the specified *classpnp* device.

```
!storagekd.storclass [Address [Level]]
```

Parameters

Address
Specifies the address to the device object or device extension of a classpnp device. If *Address* is omitted, a list of all classpnp extensions is displayed.

Level
Specifies the amount of detail to display. This parameter can be set to 0, 1, or 2, with 2 giving the most detail and 0 the least. The default is 0.

DLL

Windows 8 and later	Storagekd.dll

Remarks

Here is an example of the **!storagekd.storclass** display:

1: kd> !storagekd.storclass

```
Storage class devices:

* !storclass fffffa80043dc060 [1,2] ST3160812AS Paging Disk
  !storclass fffffa8006581740 [1,2] Msft Virtual Disk Disk

Usage: !storclass <class device> <level [0-2]>
```

1: kd> !storagekd.storclass fffffa80043dc060 1

```
Storage class device fffffa80043dc060 with extension at fffffa80043dc1b0

Classpnp Internal Information at fffffa8003bec360

    Transfer Packet Engine:

     Packet          Status  DL Irp          Opcode  Sector/ListId   UL Irp
    --------         ------ --------         ------ --------------- --------

    Pending Idle Requests: 0x0


    -- dt classpnp!_CLASS_PRIVATE_FDO_DATA fffffa8003bec360 --

Classpnp External Information at fffffa80043dc1b0

    ST3160812AS 3.ADH             9LS20QRL

    Minidriver information at fffffa80043dc670
    Attached device object at fffffa800410a060
    Physical device object at fffffa800410a060

    Media Geometry:

        Bytes in a Sector = 512
        Sectors per Track = 63
        Tracks / Cylinder = 255
        Media Length      = 160000000000 bytes = ~149 GB

    -- dt classpnp!_FUNCTIONAL_DEVICE_EXTENSION fffffa80043dc1b0 --
```

!storagekd.storhelp

12/5/2018 • 2 minutes to read • Edit Online

The **!storagekd.storhelp** extension displays help text for Storagekd.dll extension commands.

```
!storagekd.storhelp
```

DLL

Windows 8 and later	Storagekd.dll

Remarks

Here is an example of the **!storagekd.storhelp** display:

0: kd> !storagekd.storhelp

```
# Storage Debugger Extension
================================================
## General Commands
----------------
!storhelp    - Displays complete help of the commands provided in this KD extension
!storclass   - Dumps all class devices managed by classpnp
!storadapter - Dumps all adapters managed by Storport
!storunit    - Dumps all disks managed by Storport

## STORPORT specific commands
--------------------------
!storlogirp <args>     - displays internal log entries that reference the specified IRP.
                         See '!storhelp storlogirp' for details.
!storloglist <args>    - displays internal log entries. See '!storhelp storloglist' for details.
!storlogsrb <args>     - displays internal log entries that reference the specified SRB.
                         See '!storhelp storlogsrb' for details.
!storsrb <address>     - display details for the specified SCSI or STORAGE request block
```

!storagekd.storlogirp

12/5/2018 • 2 minutes to read • Edit Online

The **!storagekd.storlogirp** extension displays the Storport's internal log entries for the adapter filtered for the IRP provided.

```
!storagekd.storlogirp <Address> <irp> [<starting_entry> [<ending_entry>]] [L <count>]
```

Parameters

Address
Specifies the address of a Storport adapter device extension or device object.

irp
The IRP to locate.

starting_entry
The beginning entry in the range to display. If not specified, the last *count* entries will be displayed.

ending_entry
The ending entry in the range to display. If not specified, *count* entries will be displayed, beginning with the item specified by *starting_entry*.

count
Count of entries to be displayed. If not specified, a value of 50 is used.

DLL

Windows 8 and later	Storagekd.dll

!storagekd.storloglist

12/5/2018 • 2 minutes to read • Edit Online

The **!storagekd.storloglist** extension displays the Storport adapter's internal log entries.

```
!storagekd.storloglist <Address> [<starting_entry> [<ending_entry>]] [L <count>]
```

Parameters

Address
Specifies the address of a Storport adapter device extension or device object.

starting_entry
The beginning entry in the range to display. If not specified, the last *count* entries will be displayed.

ending_entry
The ending entry in the range to display. If not specified, *count* entries will be displayed, beginning with the item specified by *starting_entry*.

count
Count of entries to be displayed. If not specified, a value of 50 is used.

DLL

Windows 8 and later	Storagekd.dll

Remarks

Here is an example of **!storagekd.storloglist** display:

0: kd> !storagekd.storloglist ffffe0010f5e01a0

```
Storport RaidLogList
    Circular buffer location:  0xffffe0010f5e1720
    Total logs written: 8
    Displaying entries 0 through 7
    ---------------------------------------------------------------------
    [0]_[23:04:20.521] ResumeDevice.......... Caller: storport!RaidUnitPauseTimerDpcRoutine+0x28
(fffff800`fb4d0b28), P/P/T/L: 0/0/0/0, Pause count: 0, Resumed: True
    [1]_[23:04:20.646] ResumeDevice.......... Caller: storport!RaidUnitPauseTimerDpcRoutine+0x28
(fffff800`fb4d0b28), P/P/T/L: 0/0/0/0, Pause count: 0, Resumed: True
    [2]_[23:04:20.646] SpPauseDevice......... Caller: iaStorAV!RpiPauseDevice+0x67 (fffff800`fb70554f), P/T/L:
3/0/0, Timeout: 180, Adapter: 0xffffe0010f5e01a0
    [3]_[23:04:20.646] PauseDevice........... Caller: storport!StorPortPauseDevice+0x2f6 (fffff800`fb4b52d6),
P/P/T/L: 0/3/0/0, Pause count: 1
    [4]_[23:04:20.646] SpResumeDevice........ Caller: iaStorAV!RpiResumeDevice+0x5f (fffff800`fb7055fb),
P/T/L: 3/0/0, Adapter: 0xffffe0010f5e01a0
    [5]_[23:04:20.646] ResumeDevice.......... Caller: storport!StorPortResumeDevice+0x19 (fffff800`fb4aa23f),
P/P/T/L: 0/3/0/0, Pause count: 0, Resumed: True
    [6]_[23:04:20.646] SpPauseDevice......... Caller: iaStorAV!RpiPauseDevice+0x67 (fffff800`fb70554f), P/T/L:
3/0/0, Timeout: 180, Adapter: 0xffffe0010f5e01a0
    [7]_[23:04:20.646] PauseDevice........... Caller: storport!StorPortPauseDevice+0x2f6 (fffff800`fb4b52d6),
P/P/T/L: 0/3/0/0, Pause count: 1
```

!storagekd.storlogsrb

12/5/2018 • 2 minutes to read • Edit Online

The **!storagekd.storlogsrb** extension displays the Storport's internal log entries for the adapter filtered for the Storage (or SCSI) Request Block (SRB) provided.

```
!storagekd.storlogsrb <Address> <srb> [<starting_entry> [<ending_entry>]] [L <count>]
```

Parameters

Address
Specifies the address of a Storport adapter device extension or device object.

SRB
The SRB to locate.

starting_entry
The beginning entry in the range to display. If not specified, the last *count* entries will be displayed.

ending_entry
The ending entry in the range to display. If not specified, *count* entries will be displayed, beginning with the item specified by *starting_entry*.

count
Count of entries to be displayed. If not specified, a value of 50 is used.

DLL

Windows 8 and later	Storagekd.dll

!storagekd.storsrb

12/5/2018 • 2 minutes to read • Edit Online

The **!storagekd.storsrb** extension displays information about the specified Storage (or SCSI) Request Block (SRB).

```
!storagekd.storsrb Address
```

Parameters

Address
Specifies the address of the SRB.

DLL

Windows 8 and later	Storagekd.dll

Remarks

Here is an example of the **!storagekd.storsrb** display:

0: kd> !storagekd.storsrb ffffe00111fe25b0

```
    SRB is a STORAGE request block (SRB_EX)
    SRB EX 0xffffe00111fe25b0 Function 28  Version  1, Signature 53524258, SrbStatus: 0x02[Aborted],
SrbFunction 0x00 [EXECUTE SCSI]
    Address Type is BTL8

       SRB_EX Data Type [SrbExDataTypeScsiCdb16]
    [EXECUTE SCSI] SRB_EX: 0xffffe00111fe2648  OriginalRequest: 0xffffe001125a9010  DataBuffer/Length:
0xffffe00112944000 / 0x00000200
    PTL: (0, 1, 1)  CDB: 28 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00  OpCode: SCSI/READ (10)
```

!storagekd.storunit

12/5/2018 • 2 minutes to read • Edit Online

The **!storagekd.storunit** extension displays information about the specified Storport logical unit.

```
!storagekd.storunit [Address]
```

Parameters

Address
Specifies the address of a Storport unit device object. If *Address* is omitted, a list of all Storport units are displayed.

DLL

Windows 8 and later	Storagekd.dll

Remarks

Here is an example of the **!storagekd.storunit** display:

0: kd> !storagekd.storunit

```
# STORPORT Units:
==================
## Product                SCSI ID  Object            Extension         Pnd Out Ct  State
-------------------------------------------------------------------------------------
Msft       Virtual Di   0  0  1  fffffa800658a060  fffffa800658a1b0    0   0  0  Working
```

0: kd> !storagekd.storunit fffffa800658a060

```
   DO fffffa800658a060   Ext fffffa800658a1b0   Adapter fffffa800649a1a0   Working
   Vendor: Msft       Product: Virtual Disk       SCSI ID: (0, 0, 1)
   Claimed Enumerated
   SlowLock Free   RemLock 1   PageCount 0
   QueueTagList: fffffa800658a270      Outstanding: Head fffffa800658a398  Tail fffffa800658a398  Timeout -1
   DeviceQueue fffffa800658a2a0   Depth: 250   Status: Not Frozen   PauseCount: 0   BusyCount: 0
   IO Gateway: Busy Count 0   Pause Count 0
   Requests: Outstanding 0   Device 0   ByPass 0


[Device-Queued Requests]

## IRP               SRB Type   SRB               XRB               Command           MDL               SGList
Timeout
-------------------------------------------------------------------------------------------------------------------
---------------------


[Bypass-Queued Requests]

## IRP               SRB Type   SRB               XRB               Command           MDL               SGList
Timeout
-------------------------------------------------------------------------------------------------------------------
---------------------


[Outstanding Requests]

## IRP               SRB Type   SRB               XRB               Command           MDL               SGList
Timeout
-------------------------------------------------------------------------------------------------------------------
---------------------


[Completed Requests]

IRP               SRB Type   SRB               XRB               Command           MDL               SGList
Timeout
-------------------------------------------------------------------------------------------------------------------
---------------------
```

Bluetooth Extensions (Bthkd.dll)

12/5/2018 • 2 minutes to read • Edit Online

The Bluetooth debugger extensions display information about the current Bluetooth environment on the target system.

Note As you work with the Bluetooth debugging extensions, you may come across undocumented behavior or APIs. We strongly recommend against taking dependencies on undocumented behavior or APIs as it's subject to change in future releases.

In this section

TOPIC	DESCRIPTION
!bthkd.bthdevinfo	The **!bthkd.bthdevinfo** command displays the information about a given BTHENUM created device PDO.
!bthkd.bthenuminfo	The **!bthkd.bthenuminfo** command displays information about the BTHENUM FDO.
!bthkd.bthinfo	The **!bthkd.bthinfo** command displays details about the BTHPORT FDO. This command is a good starting point for Bluetooth investigations as it displays address information that can be used to access many of the other Bluetooth debug extension commands.
!bthkd.bthhelp	The **!bthkd.bthhelp** command displays help for the Bluetooth debug extension commands.
!bthkd.bthtree	The **!bthkd.bthtree** command displays the complete Bluetooth device tree.
!bthkd.bthusbtransfer	The **!bthkd.bthusbtransfer** command displays the Bluetooth usb transfer context including Irp, Bip and transfer buffer information.
!bthkd.dibflags	The **!bthkd.dibflags** command displays DEVICE_INFO_BLOCK.DibFlags dumps flags set in _DEVICE_INFO_BLOCK.DibFlags.
!bthkd.hcicmd	The **!bthkd.hcicmd** command displays a list of the currently pending commands.
!bthkd.hciinterface	The **!bthkd.hciinterface** command displays the bthport!_HCI_INTERFACE structure.

TOPIC	DESCRIPTION
!bthkd.l2capinterface	The **!bthkd.l2capinterface** command displays information about the L2CAP interface.
!bthkd.rfcomminfo	The **!bthkd.rfcomminfo** command displays information about the RFCOMM FDO and the TDI Device Object.
!bthkd.rfcommconnection	The **!bthkd.rfcommconnection** command displays information about a given RFCOMM connection object.
!bthkd.rfcommchannel	The **!bthkd.rfcommchannel** command displays information about a given RFCOMM channel CB.
!bthkd.sdpinterface	The **!bthkd.sdpinterface** command displays information about the SDP interface.
!bthkd.scointerface	The **!bthkd.scointerface** command displays information about the SCO interface.
!bthkd.sdpnode	The **!bthkd.sdpnode** command displays information about a node in an sdp tree.
!bthkd.sdpstream	The **!bthkd.sdpstream** command displays the contents of a SDP stream.

!bthkd.bthdevinfo

12/5/2018 • 2 minutes to read • Edit Online

The **!bthkd.bthdevinfo** command displays the information about a given BTHENUM created device PDO.

```
!bthkd.bthdevinfo addr
```

Parameters

addr
The address of a BTHENUM created device PDO extension.

DLL

Bthkd.dll

See also

Bluetooth Extensions (Bthkd.dll)

!bthkd.bthenuminfo

12/5/2018 • 2 minutes to read • Edit Online

The **!bthkd.bthenuminfo** command displays information about the BTHENUM FDO.

```
!bthkd.bthenuminfo
```

DLL

Bthkd.dll

See also

Bluetooth Extensions (Bthkd.dll)

!bthkd.bthinfo

12/5/2018 • 2 minutes to read • Edit Online

The **!bthkd.bthinfo** command displays details about the BTHPORT FDO. This command is a good starting point for Bluetooth investigations as it displays address information that can be used to access many of the other Bluetooth debug extension commands.

```
!bthkd.bthinfo
```

DLL

Bthkd.dll

See also

Bluetooth Extensions (Bthkd.dll)

!bthkd.bthhelp

12/5/2018 • 2 minutes to read • Edit Online

The **!bthkd.bthhelp** command displays help for the Bluetooth debug extension commands.

```
!bthkd.bthhelp
```

DLL

Bthkd.dll

See also

Bluetooth Extensions (Bthkd.dll)

!bthkd.bthtree

12/5/2018 • 2 minutes to read • Edit Online

The **!bthkd.bthtree** command displays the complete Bluetooth device tree.

```
!bthkd.bthtree
```

DLL

Bthkd.dll

See also

Bluetooth Extensions (Bthkd.dll)

!bthkd.bthusbtransfer

12/5/2018 • 2 minutes to read • Edit Online

The **!bthkd.bthusbtransfer** command displays the Bluetooth usb transfer context including Irp, Bip and transfer buffer information.

```
!bthkd.bthusbtransfer addr
```

Parameters

addr
Address of the Bluetooth USB transfer context.

Remarks

You can use the !bthinfo command to display the address of USB transfer context. It is listed under the transfer list section.

DLL

Bthkd.dll

See also

Bluetooth Extensions (Bthkd.dll)

!bthkd.dibflags

12/5/2018 • 2 minutes to read • Edit Online

The **!bthkd.dibflags** command displays DEVICE_INFO_BLOCK.DibFlags dumps flags set in _DEVICE_INFO_BLOCK.DibFlags.

```
!bthkd.dibflags flags
```

Parameters

flags
The value of _DEVICE_INFO_BLOCK.DibFlags Dumps Flags set in _DEVICE_INFO_BLOCK.DibFlags.

DLL

Bthkd.dll

See also

Bluetooth Extensions (Bthkd.dll)

!bthkd.hcicmd

12/5/2018 • 2 minutes to read • Edit Online

The **!bthkd.hcicmd** command displays a list of the currently pending commands.

```
!bthkd.hcicmd
```

DLL

Bthkd.dll

See also

Bluetooth Extensions (Bthkd.dll)

!bthkd.hciinterface

12/5/2018 • 2 minutes to read • Edit Online

The **!bthkd.hciinterface** command displays the bthport!_HCI_INTERFACE structure.

```
!bthkd.hciinterface
```

DLL

Bthkd.dll

See also

Bluetooth Extensions (Bthkd.dll)

!bthkd.l2capinterface

12/5/2018 • 2 minutes to read • Edit Online

The **!bthkd.l2capinterface** command displays information about the L2CAP interface.

```
!bthkd.l2capinterface
```

DLL

Bthkd.dll

See also

Bluetooth Extensions (Bthkd.dll)

!bthkd.rfcomminfo

12/5/2018 • 2 minutes to read • Edit Online

The **!bthkd.rfcomminfo** command displays information about the RFCOMM FDO and the TDI Device Object.

```
!bthkd.rfcomminfo
```

DLL

Bthkd.dll

See also

Bluetooth Extensions (Bthkd.dll)

!bthkd.rfcommconnection

12/5/2018 • 2 minutes to read • Edit Online

The **!bthkd.rfcommconnection** command displays information about a given RFCOMM connection object.

```
!bthkd.rfcommconnection addr
```

Parameters

addr
The address of a rfcomm!_RFCOMM_CONN_OBJ structure.

DLL

Bthkd.dll

See also

Bluetooth Extensions (Bthkd.dll)

!bthkd.rfcommchannel

12/5/2018 • 2 minutes to read • Edit Online

The **!bthkd.rfcommchannel** command displays information about a given RFCOMM channel CB.

```
!bthkd.rfcommchannel addr
```

Parameters

addr
The address of a rfcomm!_CHANNEL_CB structure.

DLL

Bthkd.dll

See also

Bluetooth Extensions (Bthkd.dll)

!bthkd.sdpinterface

12/5/2018 • 2 minutes to read • Edit Online

The **!bthkd.sdpinterface** command displays information about the SDP interface.

```
!bthkd.sdpinterface
```

DLL

Bthkd.dll

See also

Bluetooth Extensions (Bthkd.dll)

!bthkd.scointerface

12/5/2018 • 2 minutes to read • Edit Online

The **!bthkd.scointerface** command displays information about the SCO interface.

```
!bthkd.scointerface
```

DLL

Bthkd.dll

See also

Bluetooth Extensions (Bthkd.dll)

!bthkd.sdpnode

12/5/2018 • 2 minutes to read • Edit Online

The **!bthkd.sdpnode** command displays information about a node in an sdp tree.

```
!bthkd.sdpnode addr [flags]
```

Parameters

addr
Address of the sdp tree node to display.

flags
0x1 - Recurse node

0x2 - Verbose

default is 0x0

DLL

Bthkd.dll

See also

Bluetooth Extensions (Bthkd.dll)

!bthkd.sdpstream

12/5/2018 • 2 minutes to read • Edit Online

The **!bthkd.sdpstream** command displays the contents of a SDP stream.

```
!bthkd.sdpstream streamaddr streamlength
```

Parameters

streamaddr
A pointer to beginning of stream.

streamlength
The length of SDP stream in bytes to display.

DLL

Bthkd.dll

See also

Bluetooth Extensions (Bthkd.dll)

GPIO Extensions

12/5/2018 • 2 minutes to read • Edit Online

The General Purpose Input/Output (GPIO) extension commands display the software state of GPIO controllers. These commands display information from data structures maintained by the GPIO framework extension driver (Msgpioclx.sys). For information about the GPIO framework extension, see General-Purpose I/O (GPIO) Drivers.

The GPIO debugger extension commands are implemented in gpiokd.dll. To load the GPIO commands, enter **.load gpiokd.dll** in the debugger.

Each GPIO controller has a set of banks. Each bank has a pin table that has an array of pins. The GPIO debugger extension commands display information about GPIO controllers, banks, pin tables, and pins.

Data structures used by the GPIO commands

The GPIO debugger extension commands use these data structures, which are defined by Msgpioclx.sys.

msgpioclx!_DEVICE_EXTENSION
The device extension structure for the GPIO framework extension driver. This structure holds information about an individual GPIO controller.

msgpioclx!_GPIO_BANK_ENTRY
This structure holds information about an individual bank of a GPIO controller.

msgpioclx!_GPIO_PIN_INFORMATION_ENTRY
This structure holds information about an individual pin in a bank of a GPIO controller.

Getting started with GPIO debugging

To start debugging a GPIO issue, enter the **!gpiokd.clientlist** command. The **!gpiokd.clientlist** command displays an overview of all registered GPIO controllers and displays addresses that you can pass to other GPIO debugger commands.

In this section

TOPIC	DESCRIPTION
!gpiokd.help	The **!gpiokd.help** command displays help for the GPIO debugger extension commands.
!gpiokd.bankinfo	The **!gpiokd.bankinfo** command displays information about a GPIO bank.
!gpiokd.clientlist	The **!gpiokd.clientlist** command displays all registered GPIO controllers.
!gpiokd.gpioext	The **!gpiokd.gpioext** command displays information about a GPIO controller.

TOPIC	DESCRIPTION
!gpiokd.pininfo	The **!gpiokd.pininfo** command displays information about a specified GPIO pin.
!gpiokd.pinisrvec	The **!gpiokd.pinisrvec** command displays Interrupt Service Routine (ISR) vector information for a specified pin.
!gpiokd.pintable	The **!gpiokd.pintable** command displays information about an array of GPIO pins.

Related topics

Specialized Extension Commands

!gpiokd.help

12/5/2018 • 2 minutes to read • Edit Online

The **!gpiokd.help** command displays help for the GPIO debugger extension commands.

DLL

Gpiokd.dll

See also

GPIO Extensions

!gpiokd.bankinfo

12/5/2018 • 2 minutes to read • Edit Online

The **!gpiokd.bankinfo** command displays information about a GPIO bank.

```
!gpiokd.bankinfo BankAddress [Flags]
```

Parameters

BankAddress
Address of a _GPIO_BANK_ENTRY structure that represents a bank of a GPIO controller.

Flags
Flags that specify which information is displayed. This parameter is a bitwise OR of one or more of the following flags.

FLAG	DESCRIPTION
0x1	Displays the pin table.
0x2	Displays the Enable and Mask registers.
0x4	If bit 0 (0x1) is set and this flag (0x4) is set, the display includes unconfigured pins.

DLL

Gpiokd.dll

See also

GPIO Extensions

!gpiokd.clientlist

12/5/2018 • 2 minutes to read • Edit Online

The **!gpiokd.clientlist** command displays all registered GPIO controllers.

```
!gpiokd.clientlist [Flags]
```

Parameters

Flags
Flags that specify which information is displayed. This parameter is a bitwise OR of one or more of the following flags.

FLAG	DESCRIPTION
0x1	For each controller, displays detailed information including all of its banks.
0x2	If bit 0 (0x1) is set and this flag (0x2) is set, displays the Enable and Mask registers for each bank.
0x4	If bit 0 (0x1) is set and this flag (0x4) is set, the display includes unconfigured pins.

DLL

Gpiokd.dll

See also

GPIO Extensions

!gpiokd.gpioext

12/5/2018 • 2 minutes to read • Edit Online

The **!gpiokd.gpioext** command displays information about a GPIO controller.

```
!gpiokd.gpioext ExtensionAddress [Flags]
```

Parameters

ExtensionAddress
Address of the _DEVICE_EXTENSION structure that represents the GPIO controller.

Flags
Flags that specify which information is displayed. This parameter is a bitwise OR of one or more of the following flags.

FLAG	DESCRIPTION
0x1	Displays the pin table for each bank.
0x2	If bit 0 (0x1) is set and this flag (0x2) is set, the display includes the Enable and Mask registers for each bank.
0x4	If bit 0 (0x1) is set and this flag (0x4) is set, the display includes unconfigured pins.

DLL

Gpiokd.dll

See also

GPIO Extensions

!gpiokd.pininfo

12/5/2018 • 2 minutes to read • Edit Online

The **!gpiokd.pininfo** command displays information about a specified GPIO pin.

```
!gpiokd.pininfo PinAddress
```

Parameters

PinAddress
Address of the _GPIO_PIN_INFORMATION_ENTRY data structure that represents the pin.

DLL

Gpiokd.dll

See also

GPIO Extensions

!gpiokd.pinisrvec

12/5/2018 • 2 minutes to read • Edit Online

The **!gpiokd.pinisrvec** command displays Interrupt Service Routine (ISR) vector information for a specified pin.

```
!gpiokd.bankinfo PinAddress
```

Parameters

PinAddress
Address of the _GPIO_PIN_INFORMATION_ENTRY data structure that represents the pin.

DLL

Gpiokd.dll

See also

GPIO Extensions

!gpiokd.pintable

12/5/2018 • 2 minutes to read • Edit Online

The **!gpiokd.pintable** command displays information about an array of GPIO pins.

```
!gpiokd.pintable PinBase PinCount [Flags]
```

Parameters

PinBase
Address of an array of _GPIO_PIN_INFORMATION_ENTRY structures.

PinCount
The number of pins to display.

Flags
Flags that specify which information is displayed. This parameter is a bitwise OR of one or more of the following flags.

FLAG	DESCRIPTION
0x1	Not used by this command.
0x2	Not used by this command.
0x4	The display includes unconfigured pins.

DLL

Gpiokd.dll

See also

GPIO Extensions

USB 3.0 Extensions

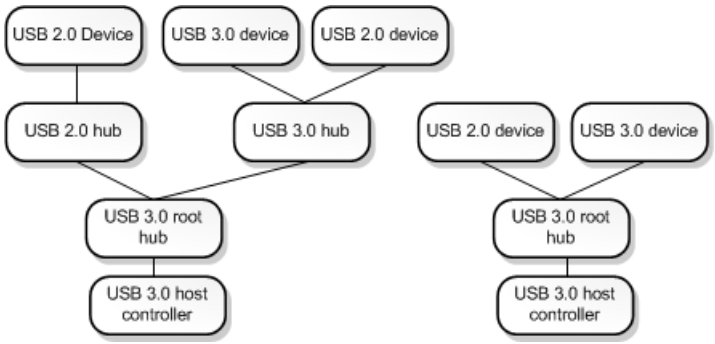
12/5/2018 • 2 minutes to read • Edit Online

This section describes the USB 3.0 debugger extension commands. These commands display information from data structures maintained by three drivers in the USB 3.0 stack: the USB 3.0 hub driver, the USB host controller extension driver, and the USB 3.0 host controller driver. For more information about these three drivers, see USB Driver Stack Architecture. For an explanation of the data structures used by the drivers in the USB 3.0 stack, see USB 3.0 Data Structures and Part 2 of the USB Debugging Innovations in Windows 8 video.

The USB 3.0 debugger extension commands are implemented in Usb3kd.dll. To load the Usb3kd commands, enter **.load usb3kd.dll** in the debugger.

USB 3.0 Tree

The USB 3.0 tree contains all USB 3.0 host controllers and all hubs and devices that are connected to USB 3.0 host controllers. The following diagram shows an example of a USB 3.0 tree.



The tree shown in the diagram has two USB 3.0 host controllers. Notice that not every device shown in the diagram is a USB 3.0 device. But all of the devices shown (including the hubs) are part of the USB 3.0 tree, because each device is on a branch that originates at a USB 3.0 host controller.

You can think of the diagram as two trees, one for each host controller. However, when we use the term *USB 3.0 tree*, we are referring to the set of all USB 3.0 host controllers along with their connected hubs and devices.

Getting started with USB 3.0 debugging

To start debugging a USB 3.0 issue, enter the **!usb_tree** command. The **!usb_tree** command displays a list of commands and addresses that you can use to investigate host controllers, hubs, ports, devices, endpoints, and other elements of the USB 3.0 tree.

Hub commands

The following extension commands display information about USB 3.0 hubs, devices, and ports. The displayed information is based on data structures maintained by the USB 3.0 hub driver.

- **!usb3kd.usb_tree**
- **!usb3kd.hub_info**
- **!usb3kd.hub_info_from_fdo**
- **!usb3kd.device_info**
- **!usb3kd.device_info_from_pdo**

- **!usb3kd.port_info**

UCX commands

The following extension commands display information about USB 3.0 host controllers, devices, and ports. The displayed information is based on data structures maintained by the USB host controller extension driver.

- **!usb3kd.ucx_controller_list**
- **!usb3kd.ucx_controller**
- **!usb3kd.ucx_device**
- **!usb3kd.ucx_endpoint**

Host controller commands

The following extension commands display information from data structures maintained by the USB 3.0 host controller driver.

- **!usb3kd.xhci_dumpall**
- **!usb3kd.xhci_capability**
- **!usb3kd.xhci_commandring**
- **!usb3kd.xhci_deviceslots**
- **!usb3kd.xhci_eventring**
- **!usb3kd.xhci_registers**
- **!usb3kd.xhci_resourceusage**
- **!usb3kd.xhci_trb**
- **!usb3kd.xhci_transferring**
- **!usb3kd.xhci_findowner**

Miscellaneous commands

- **!usb3kd.usbdstatus**
- **!usb3kd.urb**

Related topics

RCDRKD Extensions

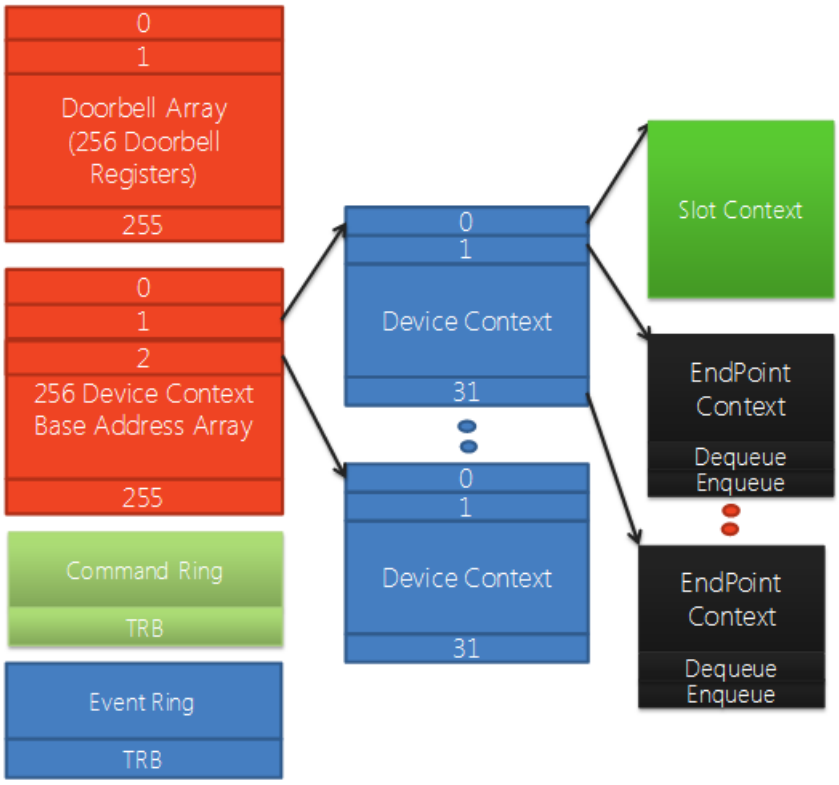
USB 3.0 Data Structures

12/5/2018 • 2 minutes to read • Edit Online

This topic describes the data structures used by the USB 3.0 host controller driver. Understanding these data structures will help you use the USB 3.0 and RCDRKD debugger extension commands effectively. The data structures presented here have names that are consistent with the USB 3.0 specification. Familiarity with the USB 3.0 specification will further enhance your ability to use the extension commands to debug USB 3.0 drivers.

The USB 3.0 host controller driver is part of the USB 3.0 core driver stack. For more information, see USB Driver Stack Architecture.

Each USB 3.0 host controller can have up to 255 devices, and each device can have up to 31 endpoints. The following diagram shows some of the data structures that represent one host controller and the connected devices.



Device Context Base Array

The Device Context Base Array is an array of pointers to Device Context structures. There is one Device Context structure for each device connected to the host controller. Elements 1 through 255 point to Device Context structures. Element 0 points to a context structure for the host controller.

Device Context and Slot Context

A Device Context structure holds an array of pointers to Endpoint Context structures. There is one Endpoint Context structure for each endpoint on the device. Elements 1 through 31 point to Endpoint Context structures. Element 0 points to a Slot Context structure, which holds context information for the device.

Command Ring

The Command Ring is used by software to pass commands to the host controller. Some of these commands are

directed at the host controller, and some are directed at particular devices connected to the host controller.

Event Ring

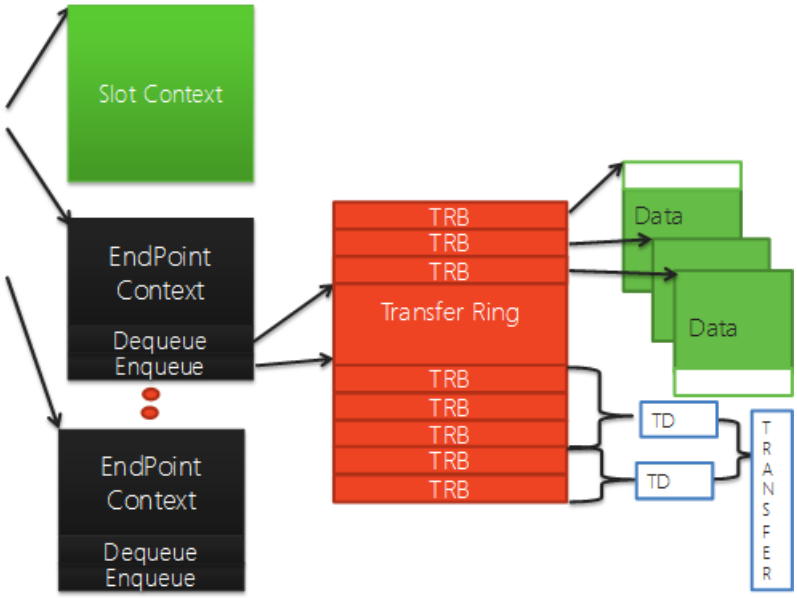
The Event Ring is used by the host controller to pass events to software. That is, the Event Ring is a structure that the host controller uses to inform drivers that an action has completed.

Doorbell Register Array

The Doorbell Register Array is an array of doorbell registers, one for each device connected to the host controller. Elements 1 through 255 are doorbell registers. Element 0 indicates whether there is a pending command in the Command Ring.

Software notifies the host controller that it has device-related or endpoint-related work to perform by writing context information into the doorbell register for the device.

The following diagram continues to the right of the preceding diagram. It shows additional data structures that represent a single endpoint.



Transfer Ring

Each endpoint has one or more Transfer Rings. A Transfer Ring is an array of Transfer Request Blocks (TRBs). Each TRB points to a block of contiguous data (up to 64 KB) that will be transferred between hardware and memory as a single unit.

When the USB 3.0 core stack receives a transfer request from a USB client driver, it identifies the Endpoint Context for the transfer, and then breaks the transfer request into one or more Transfer Descriptors (TDs). Each TD contains one or more TRBs.

Endpoint Context

An Endpoint Context structure holds context information for a single endpoint. It also has **Dequeue** and **Enqueue** members, which are used to track where TRBs are being consumed by the hardware and where TRBs are being added by software.

Related topics

!usb3kd.help

12/5/2018 • 2 minutes to read • Edit Online

The **!usb3kd.help** command displays help for the USB 3 debugger extension commands.

DLL

Usb3kd.dll

See also

USB 3.0 Extensions

!usb3kd.device_info

12/5/2018 • 2 minutes to read • Edit Online

The **!usb3kd.device_info** command displays information about a USB device in the USB 3.0 tree.

```
!usb3kd.device_info DeviceContext
```

Parameters

DeviceContext
Address of the _DEVICE_CONTEXT structure that represents the device.

DLL

Usb3kd.dll

Remarks

!device_info and **!ucx_device** both display information about a device, but the information displayed is different. The output of **!device_info** is from the point of view of the USB 3.0 hub driver, and the output of **!ucx_device** is from the point of view of the USB host controller extension driver. For example, the **!device_info** output includes information about configuration and interface descriptors, and **!ucx_device** output includes information about endpoints.

Examples

You can obtain the address of the device context structure by looking at the output of the **!usb_tree** command. In the following example, the address of the device context structure is 0xfffffa8005abd0c0.

```
3: kd> !usb_tree

## Dumping HUB Tree - !drvObj 0xfffffa800597f770
-------------------------------------------

Topology
--------
1)  !xhci_info 0xfffffa80051d1940  ... - PCI: VendorId ...
    !hub_info 0xfffffa8005ad92d0 (ROOT)
    ...
## Enumerated Device List
----------------------
...

2) !device_info 0xfffffa8005abd0c0, !devstack fffffa80059c3800
    Current Device State: ConfiguredInD0
    Desc: ... USB Flash Drive
    ...
```

Now you can pass the address of the device context to the **!device_info** command.

```
3: kd> !device_info 0fffffa8005abd0c0

## Dumping Device Information fffffa8005abd0c0
-----------------------------------------
dt USBHUB3!_DEVICE_CONTEXT 0fffffa8005abd0c0
dt USBHUB3!_HUB_PDO_CONTEXT 0fffffa8005b118d0
!idle_info 0fffffa8005b11920 (dt USBHUB3!_ISM_CONTEXT 0fffffa8005b11920)
Parent !hub_info 0fffffa8005ad92d0 (dt USBHUB3!_HUB_FDO_CONTEXT 0fffffa8005ad92d0)
!port_info 0fffffa8005abe0c0 (dt USBHUB3!_PORT_CONTEXT 0fffffa8005abe0c0)
!ucx_device 0fffffa8005992840 !xhci_deviceslots 0fffffa80051d1940 1

LPMState: U1IsEnabledForUpStreamPort U2IsEnabledForUpStreamPort U1Timeout: 38, U2Timeout: 3
DeviceFlags: HasContainerId NoBOSContainerId Removable HasSerialNumber MsOsDescriptorNotSupported
NoWakeUpSupport DeviceIsSuperSpeedCapable
DeviceHackFlags: WillDisableOnSoftRemove SkipSetIsochDelay WillResetOnResumeS0 DisableOnSoftRemove

Descriptors:
dt _USB_CONFIGURATION_DESCRIPTOR 0fffffa80053f9250
dt _USB_INTERFACE_DESCRIPTOR 0fffffa80053f9259
ProductId: ... Flash Drive
DeviceDescriptor: VID ... PID ... REV 0a00,  Class: (0)(0) BcdUsb: 0300

UcxRequest: !wdfrequest 0x57ffa662948,
ControlRequest: !wdfrequest 0x57ffa667798, !irp 0fffffa8005997650 !urb 0fffffa8005abd1c0, NumberOfBytes
0
Device working at SuperSpeed
Current Device State: ConfiguredInD0

Device State History: <Event> NewState (<Operation>(),..) :

    [16] <Yes> ConfiguredInD0
    ...
Device Event History:
    [10] TransferSuccess
    ...
```

See also

USB 3.0 Extensions

!usb3kd.device_info_from_pdo

Universal Serial Bus (USB) Drivers

!usb3kd.device_info_from_pdo

12/5/2018 • 2 minutes to read • Edit Online

The **!usb3kd.device_info_from_pdo** command displays information about a USB device in the USB 3.0 tree.

```
!usb3kd.device_info_from_pdo DeviceObject
```

Parameters

DeviceObject
Address of the physical device object (PDO) of a USB device or hub.

DLL

Usb3kd.dll

Remarks

!device_info_from_pdo and **!ucx_device** both display information about a device, but the information displayed is different. The output of **!device_info_from_pdo** is from the point of view of the USB 3.0 hub driver, and the output of **!ucx_device** is from the point of view of the USB host controller extension driver. For example, the **!device_info_from_pdo** output includes information about configuration and interface descriptors, and **!ucx_device** output includes information about endpoints.

Examples

You can get the address of the PDO from the output of **!usb_tree** or from a variety of other debugger commands. For example, the **!devnode** command displays the addresses of PDOs. In the following example, the USBSTOR device node is the direct child of the USBHUB3 node. The address of the PDO for the USBSTOR node is 0xfffffa80059c3800.

```
3: kd> !devnode 0 1 usbhub3

Dumping IopRootDeviceNode (= 0xfffffa8003609cc0)
DevNode 0xfffffa8005981730 for PDO 0xfffffa8004ffc550
  InstancePath is "USB\ROOT_HUB30\5&11db9684&0&0"
  ServiceName is "USBHUB3"
  State = DeviceNodeStarted (0x308)
  Previous State = DeviceNodeEnumerateCompletion (0x30d)

  DevNode 0xfffffa8005a546a0 for PDO 0xfffffa80059c3800
    InstancePath is "USB\VID_125F&PID_312A\0902100000000000000342192873"
    ServiceName is "USBSTOR"
    State = DeviceNodeStarted (0x308)
    Previous State = DeviceNodeEnumerateCompletion (0x30d)

    DevNode 0xfffffa8005a09730 for PDO 0xfffffa8005b3a650
      InstancePath is "USBSTOR\Disk&Ven ...
      ServiceName is "disk"
      State = DeviceNodeStarted (0x308)
      Previous State = DeviceNodeEnumerateCompletion (0x30d)
```

Now you can pass the address of the PDO to the **!usb3kd.device_info_from_pdo** command.

```
3: kd> !device_info_from_pdo 0xfffffa80059c3800

## Dumping Device Information fffffa80059c3800
-----------------------------------------
!devobj 0xfffffa8004ffc550 (Root HUB)
!device_info 0xfffffa8005abd0c0 (dt usbhub3!_DEVICE_CONTEXT 0xfffffa8005abd0c0)
dt USBHUB3!_DEVICE_CONTEXT 0xfffffa8005abd0c0
dt USBHUB3!_HUB_PDO_CONTEXT 0xfffffa8005b118d0
!idle_info 0xfffffa8005b11920 (dt USBHUB3!_ISM_CONTEXT 0xfffffa8005b11920)
Parent !hub_info 0xfffffa8005ad92d0 (dt USBHUB3!_HUB_FDO_CONTEXT 0xfffffa8005ad92d0)
!port_info 0xfffffa8005abe0c0 (dt USBHUB3!_PORT_CONTEXT 0xfffffa8005abe0c0)
!ucx_device 0xfffffa8005992840 !xhci_deviceslots 0xfffffa80051d1940 1

LPMState: U1IsEnabledForUpStreamPort U2IsEnabledForUpStreamPort U1Timeout: 38, U2Timeout: 3
DeviceFlags: HasContainerId NoBOSContainerId Removable HasSerialNumber MsOsDescriptorNotSupported
NoWakeUpSupport DeviceIsSuperSpeedCapable
DeviceHackFlags: WillDisableOnSoftRemove SkipSetIsochDelay WillResetOnResumeS0 DisableOnSoftRemove

Descriptors:
dt _USB_CONFIGURATION_DESCRIPTOR 0xfffffa80053f9250
dt _USB_INTERFACE_DESCRIPTOR 0xfffffa80053f9259
ProductId: ...
DeviceDescriptor: VID ...

UcxRequest: !wdfrequest 0x57ffa662948,
ControlRequest: !wdfrequest 0x57ffa667798, !irp 0xfffffa8005997650 !urb 0xfffffa8005abd1c0, NumberOfBytes 0
Device working at SuperSpeed
Current Device State: ConfiguredInD0

Device State History: <Event> NewState (<Operation>(),..) :

    [16] <Yes> ConfiguredInD0
    ...
Device Event History:
    [10] TransferSuccess
    ...
```

The following example shows some of the output of the **!usb_tree** command. You can see the address of the PDO of one of the child device nodes as the argument to the **!devstack** command. (**!devstack fffffa80059c3800**)

```
3: kd> !usb_tree

## Dumping HUB Tree - !drvObj 0xfffffa800597f770
-----------------------------------------------

Topology
--------
1)  !xhci_info 0xfffffa80051d1940  ... - PCI: VendorId ...
    !hub_info 0xfffffa8005ad92d0 (ROOT)
        !port_info 0xfffffa8005a5ca80 !device_info 0xfffffa8005b410c0 Desc: <none> Speed: High
        ...
## Enumerated Device List
----------------------
...

2) !device_info 0xfffffa8005abd0c0, !devstack fffffa80059c3800
    Current Device State: ConfiguredInD0
    Desc: ... Flash Drive
    USB\VID_...
    !ucx_device 0xfffffa8005992840 !xhci_deviceslots 0xfffffa80051d1940 1
```

See also

USB 3.0 Extensions

!usb3kd.device_info

Universal Serial Bus (USB) Drivers

!usb3kd.dsf

12/5/2018 • 2 minutes to read • Edit Online

The **!usb3kd.dsf** extension is a toggle command that sets the debugger context to debug the DSF host driver.

```
!usb3kd.dsf
```

DLL

Usb3kd.dll

See also

USB 3.0 Extensions

Universal Serial Bus (USB) Drivers

!usb3kd.hub_info

12/5/2018 • 2 minutes to read • Edit Online

The **!usb3kd.device_info** command displays information about a hub in the USB 3.0 tree.

```
!usb3kd.hub_info DeviceExtension
```

Parameters

DeviceExtension
Address of the device extension for the hub's functional device object (FDO).

DLL

Usb3kd.dll

Examples

To obtain the address of the device extension, look at the output of the **!usb_tree** command. In the following example, the address of the device extension for the root hub is 0xfffffa8005ad92d0.

```
3: kd> !usb_tree

## Dumping HUB Tree - !drvObj 0xfffffa800597f770
-------------------------------------------
Topology
--------
1)  !xhci_info 0xfffffa80051d1940  ... - PCI: VendorId 0x1033 ...
    !hub_info 0xfffffa8005ad92d0 (ROOT)
        !port_info 0xfffffa8005a5ca80 !device_info 0xfffffa8005b410c0 Desc: <none> Speed: High
        ...
```

Now you can pass the address of the device extension to the **!hub_info** command.

```
3: kd> !hub_info fffffa8005ad92d0

## Dumping HUB Information fffffa8005ad92d0
---------------------------------------
dt USBHUB3!_HUB_FDO_CONTEXT 0xfffffa8005ad92d0
!rcdrlogdump usbhub3 -a 0xfffffa8005ad8010
Capabilities: Initialized, Configured, HasOverCurrentProtection, ...

Total number of ports: 4, HUB depth is 0
Number of 3.0 ports: 2 (Range 1 - 2)
Number of 2.0 ports: 2 (Range 3 - 4)

Descriptors:
    dt _USB_HUB_DESCRIPTOR 0xfffffa8005ad9728
    dt _USB_30_HUB_DESCRIPTOR 0xfffffa8005ad9728
    dt _USB_DEVICE_DESCRIPTOR 0xfffffa8005ad92d0
    dt _USB_CONFIGURATION_DESCRIPTOR 0xfffffa8005ad9770

List of PortContext: 4
    [1] !port_info 0xfffffa8005a5ca80    !device_info 0xfffffa8005b410c0
            Last Port Status(2.0): Connected Enabled Powered HighSpeedDeviceAttached
            Last Port Change: <none>
            Current Port(2.0) State: ConnectedEnabled.WaitingForPortChangeEvent
            Current Device State: ConfiguredInD0
    ...

Current Hub State: ConfiguredWithIntTransfer

Hub State History: <Event> NewState (<Operation>(),..) :

    [43] <OperationSuccess> ConfiguredWithIntTransfer
    ...
Hub Event History:
    [ 0] PortEnableInterruptTransfer
    ...
```

See also

USB 3.0 Extensions

!usb3kd.hub_info_from_fdo

Universal Serial Bus (USB) Drivers

!usb3kd.hub_info_from_fdo

12/5/2018 • 2 minutes to read • Edit Online

The **!usb3kd.hub_info_from_fdo** command displays information about a hub in the USB 3.0 tree.

```
!usb3kd.hub_info_from_fdo DeviceObject
```

Parameters

DeviceObject
Address of the functional device object (FDO) that represents the hub.

DLL

Usb3kd.dll

Examples

You can get the address of the FDO from the output of **!usb_tree** or from a variety of other debugger commands. For example, the **!devstack** command displays the address of the FDO. In the following example, the address of the FDO is fffffa800597a660.

```
3: kd> !devnode 0 1 usbhub3
Dumping IopRootDeviceNode (= 0xfffffa8003609cc0)
DevNode 0xfffffa8005981730 for PDO 0xfffffa8004ffc550
...
3: kd> !devstack 0xfffffa8004ffc550
  !DevObj   !DrvObj            !DevExt   ObjectName
  fffffa800597a660  \Driver\USBHUB3    fffffa8005ad92d0
> fffffa8004ffc550  \Driver\USBXHCI    fffffa8005251d00  USBPDO-0
...
```

Now you can pass the address of the FDO to the **!usb3kd.hub_info_from_fdo** command.

```
 3: kd> !hub_info_from_fdo fffffa800597a660

## Dumping HUB Information fffffa800597a660
---------------------------------------
dt USBHUB3!_HUB_FDO_CONTEXT 0xfffffa8005ad92d0
!rcdrlogdump usbhub3 -a 0xfffffa8005ad8010
Capabilities: Initialized, Configured, HasOverCurrentProtection, SelectiveSuspendSupportedByParentStack,
CannotWakeOnConnect, NotArmedForWake

Total number of ports: 4, HUB depth is 0
Number of 3.0 ports: 2 (Range 1 - 2)
Number of 2.0 ports: 2 (Range 3 - 4)

Descriptors:
    dt _USB_HUB_DESCRIPTOR 0xfffffa8005ad9728
    dt _USB_30_HUB_DESCRIPTOR 0xfffffa8005ad9728
    dt _USB_DEVICE_DESCRIPTOR 0xfffffa8005ad92d0
    dt _USB_CONFIGURATION_DESCRIPTOR 0xfffffa8005ad9770

List of PortContext: 4
    [1] !port_info 0xfffffa8005a5ca80    !device_info 0xfffffa8005b410c0
            Last Port Status(2.0): Connected Enabled Powered HighSpeedDeviceAttached
            Last Port Change: <none>
            Current Port(2.0) State: ConnectedEnabled.WaitingForPortChangeEvent
            Current Device State: ConfiguredInD0
         ...

Current Hub State: ConfiguredWithIntTransfer

Hub State History: <Event> NewState (<Operation>(),..) :

    [43] <OperationSuccess> ConfiguredWithIntTransfer
    ...

Hub Event History:
    [ 0] PortEnableInterruptTransfer
    ...
```

See also

USB 3.0 Extensions

!usb3kd.hub_info

Universal Serial Bus (USB) Drivers

!usb3kd.port_info

12/5/2018 • 2 minutes to read • Edit Online

The **!usb3kd.port_info** command displays information about a USB port in the USB 3.0 tree.

```
!usb3kd.port_info PortContext
```

Parameters

PortContext
Address of a _PORT_CONTEXT structure.

DLL

Usb3kd.dll

Examples

To obtain the address of the port context, look at the output of the **!usb_tree** command. In the following example, the address of a port context is 0xfffffa8005abe0c0.

```
3: kd> !usb_tree

## Dumping HUB Tree - !drvObj 0xfffffa800597f770
-----------------------------------------

Topology
--------
1)  !xhci_info 0xfffffa80051d1940  ... - PCI: VendorId ...
    !hub_info 0xfffffa8005ad92d0 (ROOT)
        ...
        !port_info 0xfffffa8005abe0c0 !device_info 0xfffffa8005abd0c0 Desc: ... USB Flash Drive Speed: Super
```

Now you can pass the address of the port context to the **!port_info** command.

```
3: kd> !port_info 0xfffffa8005abe0c0

## Dumping Port Context 0xfffffa8005abe0c0
--------------------------------------
dt USBHUB3!_PORT_CONTEXT 0xfffffa8005abe0c0
!hub_info 0xfffffa8005ad92d0 (dt _HUB_FDO_CONTEXT 0xfffffa8005ad92d0)
!device_info 0xfffffa8005abd0c0 (dt _DEVICE_CONTEXT 0xfffffa8005abd0c0)
!rcdrlogdump usbhub3 -a 0xfffffa8005abf6b0

PortNumber: 2
Last Port Status(3.0): Connected Enabled Powered
Last Port Change: <none>

CurrentPortEvent: PsmEventPortConnectChange
Current Port(3.0) State: ConnectedEnabled.WaitingForPortChangeEvent

Port(3.0) State History: <Event> NewState (<Operation>(),..) :

    [34] <Push> WaitingForPortChangeEvent
    ...
Port Event History:
    [ 8] TransferSuccess
    ...
```

See also

USB 3.0 Extensions

Universal Serial Bus (USB) Drivers

!usb3kd.ucx_device

10/23/2019 • 2 minutes to read • Edit Online

The **!usb3kd.ucx_device** extension displays information about a USB device in the USB 3.0 tree. The display is based on data structures maintained by the USB host controller extension driver (Ucx*Version*.sys).

```
!usb3kd.ucx_device UcxUsbDevicePrivContext
```

Parameters

UcxUsbDevicePrivContext
Address of the _UCXUSBDEVICE_PRIVCONTEXT structure that represents the device.

DLL

Usb3kd.dll

Remarks

The USB host controller extension driver (Ucx*Version*.sys) provides a layer of abstraction between the USB 3.0 hub driver and the USB 3.0 host controller driver. The extension driver has its own representation of host controllers, devices, and endpoints. The output the **!ucx_device** command is based on the data structures maintained by the extension driver. For more information about the USB host controller extension driver and the USB 3.0 host controller driver, see USB Driver Stack Architecture.

!ucx_device and **!device_info** both display information about a device, but the information displayed is different. The output of **!ucx_device** is from the point of view of the USB host controller extension driver, and the output of **!device_info** is from the point of view of the USB 3.0 hub driver. For example, the **!ucx_device** output includes information about endpoints, and the **!device_info** output includes information about configuration and interface descriptors.

Examples

To obtain the address of the UCX USB device private context, look at the output of the **!ucx_controller_list** command. In the following example, the address of the private context for the second device is 0xfffffa8005bd9680.

```
3: 3: kd> !ucx_controller_list

## Dumping List of UCX controller objects
-------------------------------------
[1] !ucx_controller 0xfffffa80052da050 (dt ucx01000!_UCXCONTROLLER_PRIVCONTEXT fffffa80052da050)
    !ucx_device 0xfffffa8005a41840
        .!ucx_endpoint 0xfffffa800533f3d0 [Blk In ], UcxEndpointStateEnabled
        ...
    !ucx_device 0xfffffa8005bd9680
        .!ucx_endpoint 0xfffffa8003694860 [Blk Out], UcxEndpointStateEnabled
        ...
```

Now you can pass the address of the UCX USB private context to the **!ucx_device** command.

```
3: kd> !ucx_device 0fffffa8005bd9680

## Dumping Ucx USB Device Information fffffa8005bd9680
-------------------------------------------------
dt ucx01000!_UCXUSBDEVICE_PRIVCONTEXT 0fffffa8005bd9680
!ucx_controller 0fffffa80052da050
ParentHub: !wdfhandle 0x57ffacbce78
DefaultEndpoint: !ucx_endpoint 0fffffa8005be0550
ListOfEndpionts:
    .!ucx_endpoint 0fffffa8003694860 [Blk Out], UcxEndpointStateEnabled
    .!ucx_endpoint 0fffffa8003686820 [Blk In ], UcxEndpointStateEnabled
    .!ucx_endpoint 0fffffa8005be0550 [Control], UcxEndpointStateEnabled
    .!ucx_endpoint 0fffffa8003695580 [Blk In ], UcxEndpointStateStale
    .!ucx_endpoint 0fffffa80036a20c0 [Blk Out], UcxEndpointStateStale

EventCallbacks:
    EvtUsbDeviceEndpointsConfigure: (0fffff880044d1164) USBXHCI!UsbDevice_UcxEvtEndpointsConfigure
    EvtUsbDeviceEnable: (0fffff880044cffac) USBXHCI!UsbDevice_UcxEvtEnable
    EvtUsbDeviceDisable: (0fffff880044d1cbc) USBXHCI!UsbDevice_UcxEvtDisable
    EvtUsbDeviceReset: (0fffff880044d2178) USBXHCI!UsbDevice_UcxEvtReset
    EvtUsbDeviceAddress: (0fffff880044d0934) USBXHCI!UsbDevice_UcxEvtAddress
    EvtUsbDeviceUpdate: (0fffff880044d0c80) USBXHCI!UsbDevice_UcxEvtUpdate
    EvtUsbDeviceDefaultEndpointAdd: (0fffff880044ede1c) USBXHCI!Endpoint_UcxEvtUsbDeviceDefaultEndpointAdd
    EvtUsbDeviceEndpointAdd: (0fffff880044edfc8) USBXHCI!Endpoint_UcxEvtUsbDeviceEndpointAdd
```

See also

USB 3.0 Extensions

!usb3kd.ucx_controller_list

Universal Serial Bus (USB) Drivers

!usb3kd.ucx_endpoint

10/23/2019 • 2 minutes to read • Edit Online

The **!usb3kd.ucx_endpoint** command displays information about an endpoint on a USB device in the USB 3.0 tree. The display is based on data structures maintained by the USB host controller extension driver (Ucx*Version*.sys).

```
!usb3kd.ucx_endpoint UcxEndpointPrivContext
```

Parameters

UcxEndpointPrivContext
Address of the _UCXENDPOINT_PRIVCONTEXT structure that represents the endpoint.

DLL

Usb3kd.dll

Remarks

The USB host controller extension driver (Ucx*Version*.sys) provides a layer of abstraction between the USB 3.0 hub driver and the USB 3.0 host controller driver. The extension driver has its own representation of host controllers, devices, and endpoints. The output the **!ucx_endpoint** command is based on the data structures maintained by the extension driver. For more information about the USB host controller extension driver and the USB 3.0 host controller driver, see USB Driver Stack Architecture.

Examples

To obtain the address of the UCX endpoint private context, look at the output of the **!ucx_controller_list** command. In the following example, the address of the private context for the first endpoint on the second device is 0xfffffa8003694860.

```
3: kd> !ucx_controller_list

## Dumping List of UCX controller objects
--------------------------------------
[1] !ucx_controller 0xfffffa80052da050 (dt ucx01000!_UCXCONTROLLER_PRIVCONTEXT fffffa80052da050)
    !ucx_device 0xfffffa8005a41840
        .!ucx_endpoint 0xfffffa800533f3d0 [Blk In ], UcxEndpointStateEnabled
        .!ucx_endpoint 0xfffffa80053405d0 [Blk Out], UcxEndpointStateEnabled
        .!ucx_endpoint 0xfffffa8005a3f710 [Control], UcxEndpointStateEnabled
        .!ucx_endpoint 0xfffffa8005bbe4e0 [Blk Out], UcxEndpointStateStale
        .!ucx_endpoint 0xfffffa8005ac4810 [Blk In ], UcxEndpointStateStale
    !ucx_device 0xfffffa8005bd9680
        .!ucx_endpoint 0xfffffa8003694860 [Blk Out], UcxEndpointStateEnabled
        .!ucx_endpoint 0xfffffa8003686820 [Blk In ], UcxEndpointStateEnabled
        .!ucx_endpoint 0xfffffa8005be0550 [Control], UcxEndpointStateEnabled
        .!ucx_endpoint 0xfffffa8003695580 [Blk In ], UcxEndpointStateStale
        .!ucx_endpoint 0xfffffa80036a20c0 [Blk Out], UcxEndpointStateStale
```

Now you can pass the address of the UCX endpoint private context to the **!ucx_endpoint** command.

```
3: kd> !ucx_endpoint 0xfffffa8003694860

## Dumping Ucx USB Endpoint Information fffffa8003694860
-----------------------------------------------------
dt ucx01000!_UCXENDPOINT_PRIVCONTEXT 0xfffffa8003694860
[Blk Out], UcxEndpointStateEnabled, MaxTransferSize: 4194304
Endpoint Address: 0x02
Endpoint Queue: !wdfqueue 0x57ffc969888

UcxEndpoint State History: <Event> NewState
    [ 3] <UcxEndpointEventOperationSuccess> UcxEndpointStateEnabled
    [ 2] <UcxEndpointEventYes> UcxEndpointStateCompletingPendingOperation1
    [ 1] <UcxEndpointEventEnableComplete> UcxEndpointStateIsAbleToStart2
    [ 0] <SmEngineEventStart> UcxEndpointStateCreated

UcxEndpoint Event History:
    [ 1] UcxEndpointEventEnableComplete
    [ 0] SmEngineEventStart

EventCallbacks:
    EvtEndpointPurge: (0xfffff880044ba6e8) USBXHCI!Endpoint_UcxEvtEndpointPurge
    EvtEndpointAbort: (0xfffff880044ba94c) USBXHCI!Endpoint_UcxEvtEndpointAbort
    EvtEndpointReset: (0xfffff880044bb854) USBXHCI!Endpoint_UcxEvtEndpointReset
```

See also

USB 3.0 Extensions

!usb3kd.ucx_controller_list

Universal Serial Bus (USB) Drivers

!usb3kd.ucx_controller

10/23/2019 • 2 minutes to read • Edit Online

The **!usb3kd.ucx_controller** command displays information about a USB 3.0 host controller. The display is based on data structures maintained by the USB host controller extension driver (Ucx*Version*.sys).

```
!usb3kd.ucx_controller UcxControllerPrivContext
```

Parameters

UcxControllerPrivContext
Address of the _UCXCONTROLLER_PRIVCONTEXT structure that represents the controller.

DLL

Usb3kd.dll

Remarks

The USB host controller extension driver (Ucx*Version*.sys) provides a layer of abstraction between the USB 3.0 hub driver and the USB 3.0 host controller driver. The extension driver has its own representation of host controllers, devices, and endpoints. The output the **!ucx_controller** command is based on the data structures maintained by the extension driver. For more information about the USB host controller extension driver and the USB 3.0 host controller driver, see USB Driver Stack Architecture.

Examples

To obtain the address of the UCX controller private context, look at the output of the **!ucx_controller_list** command. In the following example, the address of the private context is 0xfffffa80052da050.

```
3: kd> !ucx_controller_list

## Dumping List of UCX controller objects
--------------------------------------
[1] !ucx_controller 0xfffffa80052da050 (dt ucx01000!_UCXCONTROLLER_PRIVCONTEXT fffffa80052da050)
    !ucx_device 0xfffffa8005a41840
        .!ucx_endpoint 0xfffffa800533f3d0 [Blk In ], UcxEndpointStateEnabled
        ...
    !ucx_device 0xfffffa8005bd9680
        .!ucx_endpoint 0xfffffa8003694860 [Blk Out], UcxEndpointStateEnabled
        ...
```

Now you can pass the address of the UCX controller private context to the **!ucx_controller** command.

```
3: kd> !ucx_controller 0fffffa80052da050

## Dumping Ucx Controller Information ffffa80052da050
-------------------------------------------------
dt ucx01000!_UCXCONTROLLER_PRIVCONTEXT 0fffffa80052da050
Parent Device: !wdfdevice 0x57ffac91fd8
Controller Queues:
    Default               : !wdfqueue 0x57ffacc5fd8
    Address'0'Ownership   : !wdfqueue 0x57ffad5ad88
    DeviceManagement      : !wdfqueue 0x57ffacd6fd8
    ... pend on Ctrl Reset: !wdfqueue 0x57ffad48fd8

Controller Reset State History: <Event> NewState
    [ 2] <ControllerResetEventOperationSuccess> ControllerResetStateRHPdoInD0
    [ 1] <ControllerResetEventRHPdoEnteredD0> ControllerResetStateStopBlockingResetComplete1
    [ 0] <SmEngineEventStart> ControllerResetStateRhPdoInDx

Controller Reset Event History:
    [ 1] ControllerResetEventRHPdoEnteredD0
    [ 0] SmEngineEventStart

Root Hub PDO: !wdfdevice 0x57ffaf4daa8
Number of 2.0 Ports: 2
Number of 3.0 Ports: 2
RootHub Control !wdfqueue 0x57ffacb4798
RootHub Interrupt !wdfqueue 0x57ffad033f8, pending !wdfequest 0x57ffa5fe998

Device Tree:
    !ucx_device 0fffffa8005a41840
        .!ucx_endpoint 0fffffa800533f3d0 [Blk In ], UcxEndpointStateEnabled
        .!ucx_endpoint 0fffffa80053405d0 [Blk Out], UcxEndpointStateEnabled
        .!ucx_endpoint 0fffffa8005a3f710 [Control], UcxEndpointStateEnabled
        .!ucx_endpoint 0fffffa8005bbe4e0 [Blk Out], UcxEndpointStateStale
        .!ucx_endpoint 0fffffa8005ac4810 [Blk In ], UcxEndpointStateStale
    !ucx_device 0fffffa8005bd9680
        .!ucx_endpoint 0fffffa8003694860 [Blk Out], UcxEndpointStateEnabled
        .!ucx_endpoint 0fffffa8003686820 [Blk In ], UcxEndpointStateEnabled
        .!ucx_endpoint 0fffffa8005be0550 [Control], UcxEndpointStateEnabled
        .!ucx_endpoint 0fffffa8003695580 [Blk In ], UcxEndpointStateStale
        .!ucx_endpoint 0fffffa80036a20c0 [Blk Out], UcxEndpointStateStale
```

See also

USB 3.0 Extensions

!usb3kd.ucx_controller_list

Universal Serial Bus (USB) Drivers

!usb3kd.ucx_controller_list

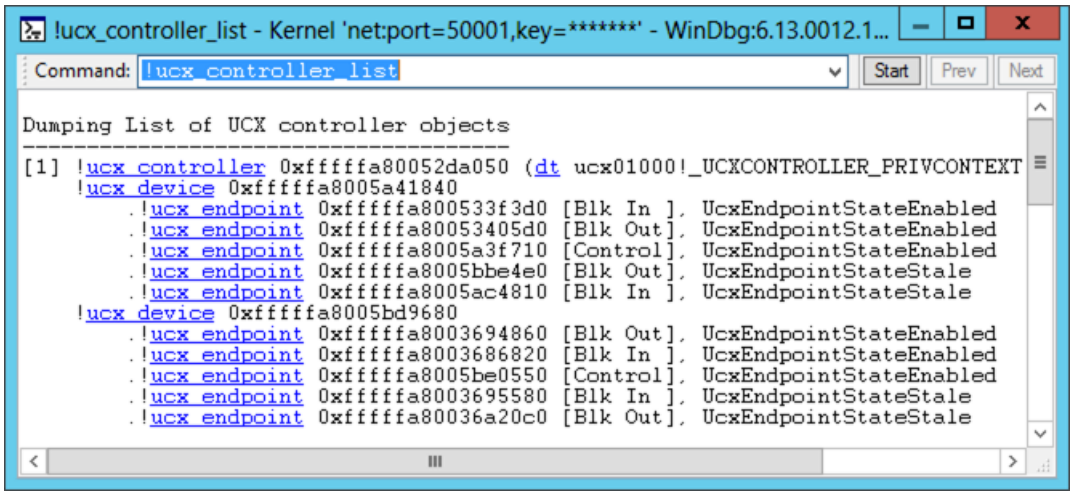
12/5/2018 • 2 minutes to read • Edit Online

The **!usb3kd.ucx_controller_list** command displays information about all USB 3.0 host controllers on the computer. The display is based on data structures maintained by the USB host controller extension driver (Ucx*Version*.sys).

```
!usb3kd.ucx_controller_list
```

Examples

The following screen shot show the output of the **!ucx_controller_list** command.



The output shows that there is one USB 3.0 host controller, which is represented by the line that begins with **!ucx_controller**. You can see that two devices are connected to the controller and that each device has four endpoints.

The output uses Using Debugger Markup Language (DML) to provide links. The links execute commands that give detailed information about individual devices or endpoints. For example, you could get detailed information about an endpoint by clicking one of the **!ucx_endpoint** links. As an alternative to clicking a link, you can enter a command. For example, to see information about the first endpoint of the second device, you could enter the command **!ucx_endpoint 0xfffffa8003694860**.

Note The DML feature is available in WinDbg, but not in Visual Studio or KD.

DLL

Usb3kd.dll

Remarks

The **!ucx_controller_list** command is the parent command for this set of commands.

- **!ucx_controller**
- **!ucx_device**
- **!ucx_endpoint**

The USB host controller extension driver (Ucx*Version*.sys) provides a layer of abstraction between the USB 3.0 hub driver and the USB 3.0 host controller driver. The extension driver has its own representation of host controllers, devices, and endpoints. The outputs of the commands in the **!ucx_controller_list** family are based on the data structures maintained by the extension driver. For more information about the USB host controller extension driver and the USB 3.0 host controller driver, see USB Driver Stack Architecture. For an explanation of the data structures used by the drivers in the USB 3.0 stack, see Part 2 of the USB Debugging Innovations in Windows 8 video.

See also

USB 3.0 Extensions

Universal Serial Bus (USB) Drivers

!usb3kd.usbanalyze

12/5/2018 • 2 minutes to read • Edit Online

The **!usb3kd.usbanalyze** extension analyzes a USB 3.0 bug check.

```
!usb3kd.usbanalyze [-v]
```

Parameters

-v
The display is verbose.

DLL

Usb3kd.dll

See also

USB 3.0 Extensions

Universal Serial Bus (USB) Drivers

BUGCODE_USB3_DRIVER

!usb3kd.usbdstatus

12/5/2018 • 2 minutes to read • Edit Online

The **!usb3kd.usbdstatus** extension displays the name of a USBD status code.

```
!usb3kd.ucx_usbdstatus UrbStatus
```

Parameters

UsbdStatus
The numeric value of a USBD status code.

DLL

Usb3kd.dll

Remarks

USBD status codes are defined in Usb.h.

Examples

The following example passes the numeric value 0x80000200 to the **!usbdstatus** command. The command returns the name of the status code, USBD_STATUS_INVALID_URB_FUNCTION.

```
3: kd> !usbdstatus 0x80000200
USBD_STATUS_INVALID_URB_FUNCTION (0x80000200)
```

See also

USB 3.0 Extensions

Universal Serial Bus (USB) Drivers

!usb3kd.usb_tree

12/5/2018 • 2 minutes to read • Edit Online

The **!usb3kd.usb_tree** extension displays information, in tree format, about all USB 3.0 controllers, hubs, and devices on the computer.

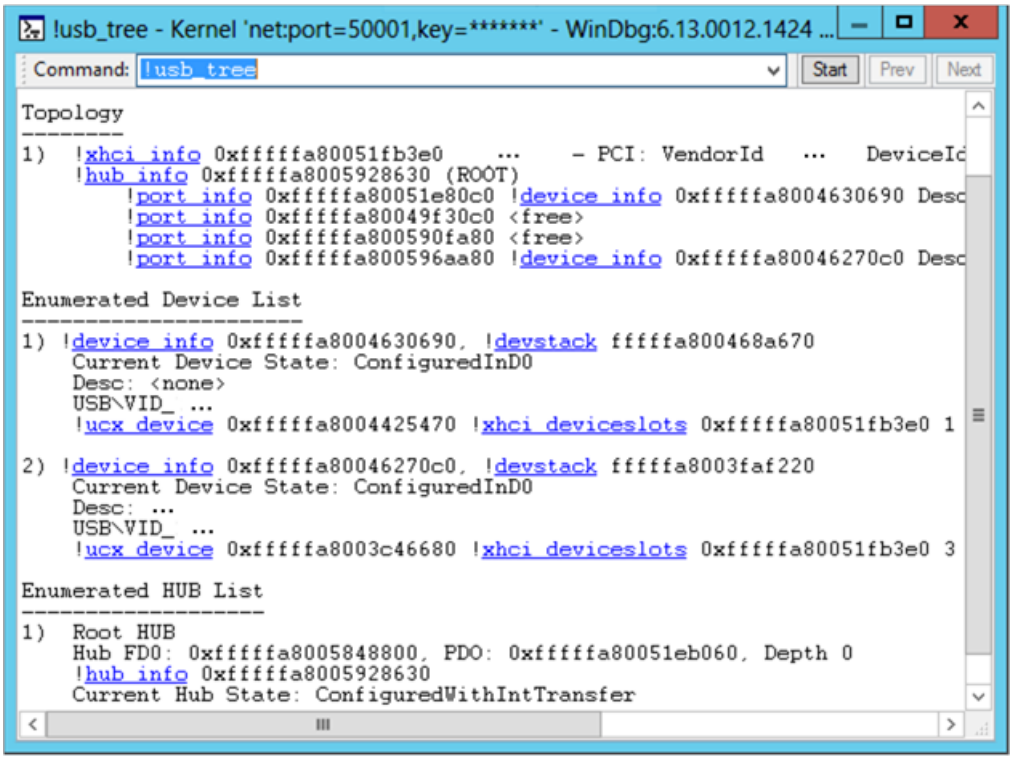
```
!usb3kd.usb_tree [1]
```

Parameters

1
The display includes status information for hubs and ports.

Examples

The following screen shot shows the output of the **!usb_tree** command.



The output shows that there is one USB 3.0 host controller, which is represented by the line that begins with **!xhci_info**. The next line represents the root hub for the host controller. The next four lines represent ports associated with the root hub. You can see that two ports have devices connected.

The output uses Using Debugger Markup Language (DML) to provide links. The links execute commands that give detailed information about individual objects in the tree. For example, you could get information about one of the connected devices by clicking one of the **!device_info** links. As an alternative to clicking a link, you can enter a command. For example, to see information about the first connected device, you could enter the command **!device_info 0xfffffa8004630690**.

Note The DML feature is available in WinDbg, but not in Visual Studio or KD.

DLL

Usb3kd.dll

Remarks

The **!usb_tree** command is the parent command for this set of commands.

- **!hub_info**
- **!hub_info_from_fdo**
- **!device_info**
- **!device_info_from_pdo**
- **!port_info**

The information displayed by the **!usb_tree** family of commands is based on data structures maintained by the USB 3.0 hub driver. For information about the USB 3.0 hub driver and other drivers in the USB 3.0 stack, see USB Driver Stack Architecture. For an explanation of the data structures used by the drivers in the USB 3.0 stack, see Part 2 of the USB Debugging Innovations in Windows 8 video.

See also

USB 3.0 Extensions

Universal Serial Bus (USB) Drivers

!usb3kd.urb

12/5/2018 • 2 minutes to read • Edit Online

The **!usb3kd.urb** extension displays information about a USB request block (URB).

```
!usb3kd.urb UrbAddress
```

Parameters

UrbAddress
Address of the URB.

DLL

Usb3kd.dll

Examples

The following example shows the address of a URB (0xfffffa8005a2cbe8) in the output of the **!xhci_deviceslots** command.

```
3: kd> !xhci_deviceslots 0xfffffa800520d2d0

## Dumping dt _DEVICESLOT_DATA 0xfffffa8003612e80
-----------------------------------------------
DeviceContextBase: VA 0xfffffa8005a64000 LA 0x116864000 !wdfcommonbuffer 0x57ffa7ca758 Size 4096

## [1] SlotID : dt USBXHCI!_USBDEVICE_DATA 0xfffffa80059027d0 dt _SLOT_CONTEXT32 0xfffffa8005a65000
-------------------------------------------------------------------------------------------------
    USB\VID_...
    SlotEnabled IsDevice NumberOfTTs 0 TTThinkTime 0
    ...
            PendingTransferList:
                [0] dt _TRANSFER_DATA 0xfffffa80059727f0 !urb 0xfffffa8005a2cbe8 !wdfrequest 0x57ffa68d998
TransferState_Pending
    ...
```

The following example passes the address of the URB to the **!usb3kd.urb** command.

```
3: kd> !urb 0fffffa8005a2cbe8

## Dumping URB 0fffffa8005a2cbe8
------------------------------
Function:         URB_FUNCTION_BULK_OR_INTERRUPT_TRANSFER (0x9)
UsbdDeviceHandle:
    !ucx_device 0fffffa8005901840
    !xhci_deviceslots 0fffffa800520d2d0 1

Status:           USBD_STATUS_PENDING (0x40000000)
UsbdFlags:        (0x0)
dt _URB_BULK_OR_INTERRUPT_TRANSFER 0fffffa8005a2cbe8
PipeHandle:            0fffffa800596f720
TransferFlags:         (0x1) USBD_TRANSFER_DIRECTION_IN
TransferBufferLength:  0x0
TransferBuffer:        0fffffa8005a2cc88
TransferBufferMDL:     0fffffa8005848930
```

See also

USB 3.0 Extensions

Universal Serial Bus (USB) Drivers

!usb3kd.xhci_capability

10/23/2019 • 2 minutes to read • Edit Online

The **!usb3kd.xhci_capability** extension displays the capabilities of a USB 3.0 host controller.

```
!usb3kd.xhci_capability DeviceExtension
```

Parameters

DeviceExtension
Address of the device extension for the host controller's functional device object (FDO).

DLL

Usb3kd.dll

Remarks

The output the **!xhci_capability** command is based on the data structures maintained by the USB 3.0 host controller driver (UsbXhci.sys). For more information about the USB 3.0 host controller driver and other drivers in the USB stack, see USB Driver Stack Architecture.

Examples

To obtain the address of the device extension, look at the output of the **!xhci_dumpall** command. In the following example, the address of the device extension is 0xfffffa800536e2d0.

```
3: kd> !xhci_dumpall

## Dumping all the XHCI controllers - DrvObj 0xfffffa80053072f0
------------------------------------------------------------
1)  ... - PCI: VendorId ... DeviceId ... RevisionId ... Firmware ...

    dt USBXHCI!_CONTROLLER_DATA 0xfffffa80052f20c0
    !rcdrlogdump USBXHCI -a 0xfffffa8005068520
    !rcdrlogdump USBXHCI -a 0xfffffa8004e8b9a0 (rundown)
    !wdfdevice 0x57ffac91fd8
    !xhci_capability 0xfffffa800536e2d0
    ...
```

Now you can pass the address of the device extension to the **!xhci_capability** command.

```
3: kd> !xhci_capability 0fffffa800536e2d0

## Controller Capabilities
-----------------------
    dt USBXHCI!_REGISTER_DATA 0xfffffa8005362c00
    dt USBXHCI!_CAPABILITY_REGISTERS 0xfffff880046a8000
    MajorRevision.MinorRevision = 0.96
    Device Slots: 32
    Interrupters: 8
    Ports: 4
    IsochSchedulingThreshold: 1
    EventRingSegmentTableMax: 1 (2^ERST = 2)
    ScratchpadRestore: OFF
    MaxScratchpadBuffers: 0
    U1DeviceExitLatency: 0
    U2DeviceExitLatency: 0
    AddressingCapability: 64 bit
    BwNegotiationCapability: ON
    ContextSize: 32 bytes
    PortPowerControl: ON
    PortIndicators: OFF
    LightHCResetCapability: OFF
    LatencyToleranceMessagingCapability: ON
    NoSecondarySidSupport: TRUE
    MaximumPrimaryStreamArraySize = 4 ( 2^(MaxPSASize+1) = 32 )
    XhciExtendedCapabilities:
        [1] USB_LEGACY_SUPPORT: dt _USBLEGSUP 0xfffff880046a8500
        [2] Supported Protocol 0xfffff880046a8510, Version 3.0, Offset 1, Count 2, HighSpeedOnly OFF,
IntegratedHub OFF, HardwareLPM OFF
        [3] Supported Protocol 0xfffff880046a8520, Version 2.0, Offset 3, Count 2, HighSpeedOnly OFF,
IntegratedHub OFF, HardwareLPM OFF

## Software Supported Capabilities
-------------------------------
    DeviceSlots: 32
    Interrupters: 1
    Ports: 4
    MaxEventRingSegments: 2
    U1DeviceExitLatency: 0
    U2DeviceExitLatency: 0
    DeviceFlags:
        IgnoreBiosHandoffFailure
        SetLinkTrbChainBit
        UseSingleInterrupter
        DisableIdlePowerManagement
```

See also

USB 3.0 Extensions

!xhci_dumpall

Universal Serial Bus (USB) Drivers

!usb3kd.xhci_commandring

12/5/2018 • 2 minutes to read • Edit Online

The **!usb3kd.xhci_commandring** extension displays information about the command ring data structure associated with a USB 3.0 host controller.

```
!usb3kd.xhci_commandring DeviceExtension
```

Parameters

DeviceExtension
AAddress of the device extension for the host controller's functional device object (FDO).

DLL

Usb3kd.dll

Remarks

The output the **!xhci_commandring** command is based on the data structures maintained by the USB 3.0 host controller driver (UsbXhci.sys). For more information about the USB 3.0 host controller driver and other drivers in the USB stack, see USB Driver Stack Architecture.

The command ring is a data structure used by the USB 3.0 host controller driver to pass commands to the host controller.

Examples

To obtain the address of the device extension, look at the output of the **!xhci_dumpall** command. In the following example, the address of the device extension is 0xfffffa800536e2d0.

```
3: kd> !xhci_dumpall

## Dumping all the XHCI controllers - DrvObj 0xfffffa80053072f0
------------------------------------------------------------
1)  ... - PCI: VendorId ... DeviceId ... RevisionId ... Firmware ...

    dt USBXHCI!_CONTROLLER_DATA 0xfffffa80052f20c0
    !rcdrlogdump USBXHCI -a 0xfffffa8005068520
    !rcdrlogdump USBXHCI -a 0xfffffa8004e8b9a0 (rundown)
    !wdfdevice 0x57ffac91fd8
    !xhci_capability 0xfffffa800536e2d0
    !xhci_registers 0xfffffa800536e2d0
    !xhci_commandring 0xfffffa800536e2d0 (No commands are pending)
    ...
```

Now you can pass the address of the device extension to the **!xhci_commandring** command.

```
3: kd> !xhci_commandring 0xfffffa800536e2d0

## Dumping dt _COMMAND_DATA 0xfffffa8005362f70 !rcdrlogdump USBXHCI -a 0xfffffa8005a8f010
--------------------------------------------------------------------------------
Stop: OFF Abort: OFF Running: ON
CommandRingBufferData: VA 0xfffffa8005aeb200 LA 0x1168eb200 !wdfcommonbuffer 0x57ffa65d988 Size 512
DequeueIndex: 24 EnqueueIndex: 24 CycleState: 0

    Command Ring TRBs:
        [  0] Unknown TRB Type 49 0xfffffa8005aeb200

        [  1] ENABLE_SLOT                0xfffffa8005aeb210 CycleBit 1
        [  2] ADDRESS_DEVICE             0xfffffa8005aeb220 CycleBit 1 SlotId  1 BlockSetAddressRequest 1
        ...

    PendingList:
        Empty List

    WaitingList:
        Empty List
```

See also

USB 3.0 Extensions

!xhci_dumpall

Universal Serial Bus (USB) Drivers

!usb3kd.xhci_deviceslots

12/5/2018 • 2 minutes to read • Edit Online

The **!usb3kd.xhci_deviceslots** extension displays information about the devices connected to a USB 3.0 host controller.

```
!usb3kd.xhci_deviceslots DeviceExtension [SlotNumber] [verbose]
```

Parameters

DeviceExtension
Address of the device extension for the host controller's functional device object (FDO).

SlotNumber
Slot number of the device to be displayed. If this parameter is omitted, all devices are displayed.

verbose
The display is verbose.

DLL

Usb3kd.dll

Remarks

The output the **!xhci_deviceslots** command is based on the data structures maintained by the USB 3.0 host controller driver (UsbXhci.sys). For more information about the USB 3.0 host controller driver and other drivers in the USB stack, see USB Driver Stack Architecture.

The USB 3.0 host controller driver maintains a list of data structures that represent the devices connected to the controller. Each of these data structures is identified by a slot number.

Examples

To obtain the address of the device extension, look at the output of the **!xhci_dumpall** command. In the following example, the address of the device extension is 0xfffffa800536e2d0.

```
3: kd> !xhci_dumpall

## Dumping all the XHCI controllers - DrvObj 0xfffffa80053072f0
-------------------------------------------------------------
1)  ... - PCI: VendorId ... DeviceId ... RevisionId ... Firmware ...

    dt USBXHCI!_CONTROLLER_DATA 0xfffffa80052f20c0
    !rcdrlogdump USBXHCI -a 0xfffffa8005068520
    !rcdrlogdump USBXHCI -a 0xfffffa8004e8b9a0 (rundown)
    !wdfdevice 0x57ffac91fd8
    !xhci_capability 0xfffffa800536e2d0
    !xhci_registers 0xfffffa800536e2d0
    !xhci_commandring 0xfffffa800536e2d0 (No commands are pending)
    !xhci_deviceslots 0xfffffa800536e2d0
    ...
```

Now you can pass the address of the device extension to the **!usb3kd.xhci_deviceslots** command.

```
3: kd> !xhci_deviceslots 0fffffa800536e2d0

## Dumping dt _DEVICESLOT_DATA 0fffffa8005226220
-----------------------------------------------
DeviceContextBase: VA 0fffffa8005ab9000 LA 0x1168b9000 !wdfcommonbuffer 0x57ffa65c9b8 Size 4096

## [1] SlotID : dt USBXHCI!_USBDEVICE_DATA 0fffffa8005a427d0 dt _SLOT_CONTEXT32 0fffffa8005aba000
-------------------------------------------------------------------------------------------------
    USB\VID_125F&PID_312A ADATA Technology Co., Ltd.
    SlotEnabled IsDevice NumberOfTTs 0 TTThinkTime 0
    Speed: Super PortPathDepth: 1 PortPath: [ 2 ] DeviceAddress: 1
    !device_info_from_pdo 0fffffa8005a36800
    DeviceContextBuffer: VA 0fffffa8005aba000 LA 0x1168ba000 !wdfcommonbuffer 0x57ffa656948 Size 4096
    InputDeviceContextBuffer: VA 0fffffa8005b65000 LA 0x116965000 !wdfcommonbuffer 0x57ffa5be958 Size 4096

    [1] DeviceContextIndex : dt USBXHCI!_ENDPOINT_DATA 0fffffa8005a126f0 dt _ENDPOINT_CONTEXT32
0fffffa8005aba020 ES_RUNNING
    -------------------------------------------------------------------------------------------------------------
----
        EndpointType_Control Address: 0x0 PacketSize: 512 Interval: 0
        !ucx_endpoint 0fffffa8005a3f710
        RecorderLog: !rcdrlogdump USBXHCI -a 0fffffa8005b60010

        [0] dt _TRANSFERRING_DATA 0fffffa8005b64ec0 Events: 0x0 TransferRingState_Idle
     ...

## [2] SlotID : dt USBXHCI!_USBDEVICE_DATA 0fffffa80052de320 dt _SLOT_CONTEXT32 0fffffa8005b8b000
-------------------------------------------------------------------------------------------------
    USB\VID_18A5&PID_0304 Verbatim Americas LLC
    SlotEnabled IsDevice NumberOfTTs 0 TTThinkTime 0
    Speed: High PortPathDepth: 1 PortPath: [ 3 ] DeviceAddress: 2
    !device_info_from_pdo 0fffffa80058fb800
    DeviceContextBuffer: VA 0fffffa8005b8b000 LA 0x11698b000 !wdfcommonbuffer 0x57ffa426b18 Size 4096
    InputDeviceContextBuffer: VA 0fffffa8005b8c000 LA 0x11698c000 !wdfcommonbuffer 0x57ffadbe3c8 Size 4096

    [1] DeviceContextIndex : dt USBXHCI!_ENDPOINT_DATA 0fffffa800714b050 dt _ENDPOINT_CONTEXT32
0fffffa8005b8b020 ES_RUNNING
    -------------------------------------------------------------------------------------------------------------
----
        EndpointType_Control Address: 0x0 PacketSize: 64 Interval: 0
        !ucx_endpoint 0fffffa80036a20c0
        RecorderLog: !rcdrlogdump USBXHCI -a 0fffffa8005bd0b60

        [0] dt _TRANSFERRING_DATA 0fffffa8004ed8df0 Events: 0x0 TransferRingState_Idle
        -----------------------------------------------------------------------------
    ...
```

See also

USB 3.0 Extensions

!xhci_dumpall

Universal Serial Bus (USB) Drivers

!usb3kd.xhci_dumpall

12/5/2018 • 2 minutes to read • Edit Online

The **!usb3kd.xhci_dumpall** command displays information about all USB 3.0 host controllers on the computer. The display is based on the data structures maintained by the USB 3.0 host controller driver (UsbXhci.sys).

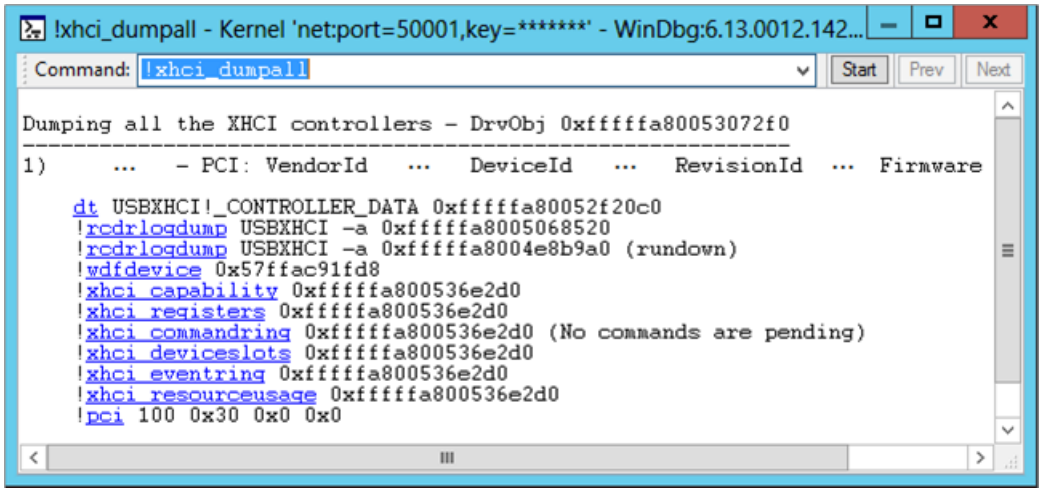
```
!usb3kd.xhci_dumpall [1]
```

Parameters

1
Runs all of the XHCI commands and displays the output of each command.

Examples

The following screen shot show the output of the **!xhci_dumpall**l command.



The output shows that there is one USB 3.0 host controller.

The output uses Using Debugger Markup Language (DML) to provide links. The links execute commands that give detailed information about the state of the host controller as it is maintained by the USB 3.0 host controller driver. For example, you could get detailed information about the host controller capabilities by clicking the **!xhci_capability** link. As an alternative to clicking a link, you can enter a command. For example, to see information about the host controller's resource usage, you could enter the command **!xhci_resourceusage 0xfffffa800536e2d0**.

Note The DML feature is available in WinDbg, but not in Visual Studio or KD.

DLL

Usb3kd.dll

Remarks

The **!xhci_dumpall** command is the parent command for this set of commands.

- **!xhci_capability**

- **!xhci_info**
- **!xhci_deviceslots**
- **!xhci_commandring**
- **!xhci_eventring**
- **!xhci_transferring**
- **!xhci_trb**
- **!xhci_registers**
- **!xhci_resourceusage**

The information displayed by the **!xhci_dumpall** family of commands is based on data structures maintained by the USB 3.0 host controller driver. For information about the USB 3.0 host controller driver and other drivers in the USB 3.0 stack, see USB Driver Stack Architecture. For an explanation of the data structures used by the drivers in the USB 3.0 stack, see Part 2 of the USB Debugging Innovations in Windows 8 video.

See also

USB 3.0 Extensions

Universal Serial Bus (USB) Drivers

!usb3kd.xhci_eventring

12/5/2018 • 2 minutes to read • Edit Online

The **!usb3kd.xhci_eventring** extension displays information about the event ring data structure associated with a USB 3.0 host controller.

```
!usb3kd.xhci_eventring DeviceExtension
```

Parameters

DeviceExtension
Address of the device extension for the host controller's functional device object (FDO).

DLL

Usb3kd.dll

Remarks

The output **!xhci_eventring** command is based on the data structures maintained by the USB 3.0 host controller driver (UsbXhci.sys). For more information about the USB 3.0 host controller driver and other drivers in the USB stack, see USB Driver Stack Architecture.

The event ring is a structure used by the USB 3.0 host controller to inform drivers that an action has completed.

Examples

To obtain the address of the device extension, look at the output of the **!xhci_dumpall** command. In the following example, the address of the device extension is 0xfffffa800536e2d0.

```
3: kd> !xhci_dumpall

## Dumping all the XHCI controllers - DrvObj 0xfffffa80053072f0
-------------------------------------------------------------
1)  ... - PCI: VendorId ... DeviceId ... RevisionId ... Firmware ...

    dt USBXHCI!_CONTROLLER_DATA 0xfffffa80052f20c0
    !rcdrlogdump USBXHCI -a 0xfffffa8005068520
    !rcdrlogdump USBXHCI -a 0xfffffa8004e8b9a0 (rundown)
    !wdfdevice 0x57ffac91fd8
    !xhci_capability 0xfffffa800536e2d0
    !xhci_registers 0xfffffa800536e2d0
    !xhci_commandring 0xfffffa800536e2d0 (No commands are pending)
    !xhci_deviceslots 0xfffffa800536e2d0
    !xhci_eventring 0xfffffa800536e2d0
    ...
```

Now you can pass the address of the device extension to the **!xhci_eventring** command.

```
3: kd> !xhci_eventring 0xfffffa800536e2d0

## Dumping dt _PRIMARY_INTERRUPTER_DATA fffffa800536b5b0
-----------------------------------------------------

## [0] Interrupter : dt _INTERRUPTER_DATA 0xfffffa800536b7d0  !rcdrlogdump USBXHCI -a 0xfffffa8005aeab60
-------------------------------------------------------------------------------------------------------
    DequeueSegment: 1 DequeueIndex: 217 TotalEventRingSegments: 2 TRBsPerSegment: 256
    CurrentBufferData : VA 0xfffffa8005373000 LA 0x117173000 !wdfcommonbuffer 0x57ffa65b9b8 Size 4096
    EventRingTableBufferData : VA 0xfffffa8005aeb000 LA 0x1168eb000 !wdfcommonbuffer 0x57ffa65d988 Size 512

    [0] VA 0xfffffa8005370000 LA 0x117170000 !wdfcommonbuffer 0x57ffa6599b8 Size 4096
    [1] VA 0xfffffa8005373000 LA 0x117173000 !wdfcommonbuffer 0x57ffa65b9b8 Size 4096

    Event Ring TRBs:
        [207] TRANSFER_EVENT      0xfffffa8005373cf0 CycleBit 0 SlotId  2 EndpointID  4 EventData 1 Pointer
0xfffffa8005366700 CC_SUCCESS
        [208] TRANSFER_EVENT      0xfffffa8005373d00 CycleBit 0 SlotId  2 EndpointID  3 EventData 1 Pointer
0xfffffa8005a3d850 CC_SHORT_PACKET
        [209] TRANSFER_EVENT      0xfffffa8005373d10 CycleBit 0 SlotId  1 EndpointID  4 EventData 1 Pointer
0xfffffa8005a3d850 CC_SUCCESS
        [210] TRANSFER_EVENT      0xfffffa8005373d20 CycleBit 0 SlotId  1 EndpointID  3 EventData 1 Pointer
0xfffffa8005366700 CC_SUCCESS
        [211] TRANSFER_EVENT      0xfffffa8005373d30 CycleBit 0 SlotId  2 EndpointID  4 EventData 1 Pointer
0xfffffa8005366700 CC_SUCCESS
        [212] TRANSFER_EVENT      0xfffffa8005373d40 CycleBit 0 SlotId  2 EndpointID  3 EventData 1 Pointer
0xfffffa8005a3d850 CC_SHORT_PACKET
        [213] TRANSFER_EVENT      0xfffffa8005373d50 CycleBit 0 SlotId  1 EndpointID  4 EventData 1 Pointer
0xfffffa8005a3d850 CC_SUCCESS
        [214] TRANSFER_EVENT      0xfffffa8005373d60 CycleBit 0 SlotId  1 EndpointID  3 EventData 1 Pointer
0xfffffa8005366700 CC_SUCCESS
        [215] TRANSFER_EVENT      0xfffffa8005373d70 CycleBit 0 SlotId  2 EndpointID  4 EventData 1 Pointer
0xfffffa8005366700 CC_SUCCESS
        [216] TRANSFER_EVENT      0xfffffa8005373d80 CycleBit 0 SlotId  2 EndpointID  3 EventData 1 Pointer
0xfffffa8005a3d850 CC_SHORT_PACKET
```

See also

USB 3.0 Extensions

!xhci_dumpall

Universal Serial Bus (USB) Drivers

!usb3kd.xhci_findowner

12/5/2018 • 3 minutes to read • Edit Online

The **!usb3kd.xhci_findowner** command finds the owner a common buffer.

```
!usb3kd.xhci_findowner Address
```

Parameters

Address
Virtual or physical address of a common buffer.

DLL

Usb3kd.dll

Remarks

A common buffer is a block of physically contiguous memory that's addressable by hardware. The USB 3.0 driver stack uses common buffers to communicate with USB 3.0 host controllers. Suppose the system crashes, and you come across an address that you suspect might be common buffer memory. If the address is common buffer memory, this command tells you which USB 3.0 host controller the memory belongs to (in case you that you have more than one USB 3.0 controller) and what the memory is used for.

Examples

The following example calls **!xhci_resourceusage** to list the addresses of some common buffers.

```
0: kd> !usb3kd.xhci_resourceusage 0x867fbff0

## Dumping CommonBuffer Resources
------------------------------
    dt USBXHCI!_COMMON_BUFFER_DATA 0x868d61c0
    DmaEnabler:!wdfdmaenabler 0x79729fe8

    CommonBuffers Large: Total 8 Available 2 Used 6 TotalBytes 32768
        [ 1] dt _TRACKING_DATA 0x868d73a4 VA 0x868e0000 LA 0xdb2e0000 -- Owner 0x86801690 Tag: Int2 Size 4096
        [ 2] dt _TRACKING_DATA 0x868d6d1c VA 0x868e1000 LA 0xdb2e1000 -- Owner 0x86801690 Tag: Int2 Size 4096
        [ 3] dt _TRACKING_DATA 0x868d6c54 VA 0x868e2000 LA 0xdb2e2000 -- Owner 0x86801690 Tag: Int2 Size 4096
        [ 4] dt _TRACKING_DATA 0x868d6b8c VA 0x868e3000 LA 0xdb2e3000 -- Owner 0x86801690 Tag: Int2 Size 4096
        [ 5] dt _TRACKING_DATA 0x868d67b4 VA 0x868e5000 LA 0xdb2e5000 -- Owner 0x86801548 Tag: Slt1 Size 4096
        [ 6] dt _TRACKING_DATA 0x868d50b4 VA 0x868e6000 LA 0xdb2e6000 -- Owner 0x86801548 Tag: Slt3 Size 4096

    CommonBuffers Small: Total 16 Available 13 Used 3 TotalBytes 8192
        [ 1] dt _TRACKING_DATA 0x868d6974 VA 0x868e4000 LA 0xdb2e4000 -- Owner 0x86801690 Tag: Int1 Size 512
        [ 2] dt _TRACKING_DATA 0x868d69a4 VA 0x868e4200 LA 0xdb2e4200 -- Owner 0x86801548 Tag: Slt2 Size 512
        [ 3] dt _TRACKING_DATA 0x868d69d4 VA 0x868e4400 LA 0xdb2e4400 -- Owner 0x86801488 Tag: Cmd1 Size 512
```

One of the virtual addresses listed in the preceding output is 0x868e2000. The following example passes that address to **!xhci_findowner**. One of the physical addresses listed in the preceding output is 0xdb2e4400. The following example passes 0xdb2e4440 (offset 0x40 bytes from 0xdb2e4400) to **!xhci_findowner**.

```
0: kd> !xhci_findowner 0x868e2000

!xhci_info 0x867fbff0  Texas Instruments - PCI: VendorId 0x104c DeviceId 0x8241 RevisionId 0x02

    dt _TRACKING_DATA 0x868d6c54 VA 0x868e2000 LA 0xdb2e2000 -- Owner 0x86801690 Tag: Int2 Size 4096

    dt _INTERRUPTER_DATA 0x86801690
    !xhci_eventring 0x867fbff0  <-- This memory is used for event ring.


0: kd> !xhci_findowner 0xdb2e4440  <-- Note the offset difference.

!xhci_info 0x867fbff0  Texas Instruments - PCI: VendorId 0x104c DeviceId 0x8241 RevisionId 0x02

    dt _TRACKING_DATA 0x868d69d4 VA 0x868e4400 LA 0xdb2e4400 -- Owner 0x86801488 Tag: Cmd1 Size 512

    dt _COMMAND_DATA 0x86801488
    !xhci_commandring 0x867fbff0  <-- This memory is used for command ring.
```

The **!xhci_findowner** command is especially useful when you have an address in a transfer request block (TRB), and you want to track it back to the device slot that it belongs to. In the following example, one of the addresses listed in the output of **!xhci_transferring** is 0xda452230, which is the physical address of a TRB. The example passes that address to **!xhci_findowner**. The output shows that the TRB belongs to device slot 8 (**!xhci_deviceslots 0x8551d370 8**).

```
0: kd> !usb3kd.xhci_transferring 0x87652200

        [  0] NORMAL       0xda452200 CycleBit 1 IOC 0 BEI 0 InterrupterTarget 2 TransferLength     6 TDSize
0
        [  1] EVENT_DATA   0xda452210 CycleBit 1 IOC 1 BEI 0 InterrupterTarget 2 Data 0x8511375c TotalBytes 6
        [  2] NORMAL       0xda452220 CycleBit 1 IOC 0 BEI 0 InterrupterTarget 2 TransferLength     6 TDSize
0
        [  3] EVENT_DATA   0xda452230 CycleBit 1 IOC 1 BEI 0 InterrupterTarget 2 Data 0x857d076c TotalBytes 6

0: kd> !xhci_findowner 0xda452230

!xhci_info 0x8551d370  Renesas - PCI: VendorId 0x1912 DeviceId 0x0015 RevisionId 0x02 Firmware 0x0020.0006

    dt _TRACKING_DATA 0x8585fd5c VA 0x87652200 LA 0xda452200 -- Owner 0x85894548 Tag: Rng1 Size 512

    !xhci_deviceslots 0x8551d370 8

    [0] dt _TRANSFERRING_DATA 0x85894548 Events: 0x0 TransferRingState_Idle
    ------------------------------------------------------------------------------
        WdfQueue: !wdfqueue 0x7a76bcb0 (0 waiting)
        CurrentRingBufferData: VA 0x87652200 LA 0xda452200 !wdfcommonbuffer 0x7a7a0370 Size 512
        Current:  !xhci_transferring 0x87652200
        PendingTransferList:
            [0] dt _TRANSFER_DATA 0x851136f0 !urb 0x84e55468 !wdfrequest 0x7aeec9e8 TransferState_Pending
            [1] dt _TRANSFER_DATA 0x857d0700 !urb 0x85733be8 !wdfrequest 0x7a82f9d8 TransferState_Pending
```

See also

USB 3.0 Extensions

!xhci_dumpall

Universal Serial Bus (USB) Drivers

!usb3kd.xhci_info

12/5/2018 • 2 minutes to read • Edit Online

The **!usb3kd.xhci_info** extension displays all the XHCI commands for an individual USB 3.0 host controller.

```
!usb3kd.xhci_info DeviceExtension
```

Parameters

DeviceExtension
Address of the device extension for the host controller's functional device object (FDO).

DLL

Usb3kd.dll

Remarks

The output the **!usb3kd.xhci_info** command is based on the data structures maintained by the USB 3.0 host controller driver (UsbXhci.sys). For more information about the USB 3.0 host controller driver and other drivers in the USB stack, see USB Driver Stack Architecture.

Examples

You can get address of the device extension from the **!xhci_dumpall** command or from a variety of other debugger commands. For example, the **!devstack** command displays the address of the device extension. In the following example, the address of the device extension for the host controller's FDO is fffffa800536e2d0.

```
3: kd> !devnode 0 1 usbxhci
Dumping IopRootDeviceNode (= 0xfffffa8003609cc0)
DevNode 0xfffffa8003df3010 for PDO 0xfffffa8003dd5870
  InstancePath is "PCI\VEN_...
  ServiceName is "USBXHCI"
  ...

3: kd> !devstack 0xfffffa8003dd5870
  !DevObj   !DrvObj            !DevExt   ObjectName
  fffffa800534b060  \Driver\USBXHCI    fffffa800536e2d0  USBFDO-3
  fffffa8003db5790  \Driver\ACPI       fffffa8003701cb0
> fffffa8003dd5870  \Driver\pci        fffffa8003dd59c0  NTPNP_PCI0020
...
```

Now you can pass the address of the device extension to the **!xhci_info** command.

```
3: kd> !xhci_info 0fffffa80`0536e2d0

## Dumping XHCI controller commands - DeviceExtension 0fffffa800536e2d0
------------------------------------------------------------
 ... - PCI: VendorId ... DeviceId ... RevisionId ... Firmware ...

    dt USBXHCI!_CONTROLLER_DATA 0fffffa80052f20c0
    !rcdrlogdump USBXHCI -a 0fffffa8005068520
    !rcdrlogdump USBXHCI -a 0fffffa8004e8b9a0 (rundown)
    !wdfdevice 0x57ffac91fd8
    !xhci_capability 0fffffa800536e2d0
    !xhci_registers 0fffffa800536e2d0
    !xhci_commandring 0fffffa800536e2d0 (No commands are pending)
    !xhci_deviceslots 0fffffa800536e2d0
    !xhci_eventring 0fffffa800536e2d0
    !xhci_resourceusage 0fffffa800536e2d0
    !pci 100 0x30 0x0 0x0
```

See also

USB 3.0 Extensions

!xhci_dumpall

Universal Serial Bus (USB) Drivers

!usb3kd.xhci_registers

12/5/2018 • 2 minutes to read • Edit Online

The **!usb3kd.xhci_registers** extension displays the registers of a USB 3.0 host controller.

```
!usb3kd.xhci_registers DeviceExtension
```

Parameters

DeviceExtension
Address of the device extension for the host controller's functional device object (FDO).

DLL

Usb3kd.dll

Remarks

The output the **!xhci_registers** command is based on the data structures maintained by the USB 3.0 host controller driver (UsbXhci.sys). For more information about the USB 3.0 host controller driver and other drivers in the USB stack, see USB Driver Stack Architecture.

Examples

To obtain the address of the device extension, look at the output of the **!xhci_dumpall** command. In the following example, the address of the device extension is 0xfffffa800536e2d0.

```
3: kd> !xhci_dumpall

## Dumping all the XHCI controllers - DrvObj 0xfffffa80053072f0
------------------------------------------------------------
1)  ... - PCI: VendorId ... DeviceId ... RevisionId ... Firmware ...

    dt USBXHCI!_CONTROLLER_DATA 0xfffffa80052f20c0
    !rcdrlogdump USBXHCI -a 0xfffffa8005068520
    !rcdrlogdump USBXHCI -a 0xfffffa8004e8b9a0 (rundown)
    !wdfdevice 0x57ffac91fd8
    !xhci_capability 0xfffffa800536e2d0
    !xhci_registers 0xfffffa800536e2d0
    ...
```

Now you can pass the address of the device extension to the **!xhci_registers** command.

```
3: kd> !xhci_registers 0fffffa800536e2d0

## Dumping controller registers
----------------------------

    dt USBXHCI!_OPERATIONAL_REGISTERS 0fffff880046a8020

    DeviceContextBaseAddressArrayPointer: 00000001168b9000

    Command Registers
    -----------------
        RunStopBit: 1
        HostControllerReset: 0
        ...

    Status Registers
    ----------------
        HcHalted: 0
        HostSystemError: 0
        ...

    commandRingControl Registers
    ----------------------------
        RingCycleState: 0
        CommandStop: 0
        ...
    Runtime Registers
    -----------------
        dt USBXHCI!_RUNTIME_REGISTERS 0fffff880046a8600
        MicroFrameIndex: 0x3f7a

        dt -ba8 USBXHCI!_INTERRUPTER_REGISTER_SET 0fffff880046a8620

    RootPort Registers
    ------------------
        dt -a4 -r2 USBXHCI!_PORT_REGISTER_SET 0fffff880046a8420
```

See also

USB 3.0 Extensions

!xhci_dumpall

Universal Serial Bus (USB) Drivers

!usb3kd.xhci_resourceusage

12/5/2018 • 2 minutes to read • Edit Online

The **!usb3kd.xhci_resourceusage** extension displays the resources used by a USB 3.0 host controller.

```
!usb3kd.xhci_resourceusage DeviceExtension
```

Parameters

DeviceExtension
Address of the device extension for the host controller's functional device object (FDO).

DLL

Usb3kd.dll

Remarks

The output the **!xhci_resourceusage** command is based on the data structures maintained by the USB 3.0 host controller driver (UsbXhci.sys). For more information about the USB 3.0 host controller driver and other drivers in the USB stack, see USB Driver Stack Architecture.

Examples

To obtain the address of the device extension, look at the output of the **!xhci_dumpall** command. In the following example, the address of the device extension is 0xfffffa800536e2d0.

```
3: kd> !xhci_dumpall

## Dumping all the XHCI controllers - DrvObj 0xfffffa80053072f0
------------------------------------------------------------
1)  ... - PCI: VendorId ... DeviceId ... RevisionId ... Firmware ...

    dt USBXHCI!_CONTROLLER_DATA 0xfffffa80052f20c0
    !rcdrlogdump USBXHCI -a 0xfffffa8005068520
    !rcdrlogdump USBXHCI -a 0xfffffa8004e8b9a0 (rundown)
    !wdfdevice 0x57ffac91fd8
    !xhci_capability 0xfffffa800536e2d0
    !xhci_registers 0xfffffa800536e2d0
    !xhci_commandring 0xfffffa800536e2d0 (No commands are pending)
    !xhci_deviceslots 0xfffffa800536e2d0
    !xhci_eventring 0xfffffa800536e2d0
    !xhci_resourceusage 0xfffffa800536e2d0
    ...
```

Now you can pass the address of the device extension to the **!xhci_resourceusage** command.

```
 3: kd> !xhci_resourceusage 0xfffffa800536e2d0

## Dumping CommonBuffer Resources
------------------------------
    dt USBXHCI!_COMMON_BUFFER_DATA 0xfffffa80059a5920
    DmaEnabler:!wdfdmaenabler 0x57ffa65a9c8

    CommonBuffers Large: Total 9 Available 2 Used 7 TotalBytes 36864
        [ 1] dt _TRACKING_DATA 0xfffffa80059a6768 VA 0xfffffa8005370000 LA 0x117170000 ...
        [ 2] dt _TRACKING_DATA 0xfffffa80059a4768 VA 0xfffffa8005373000 LA 0x117173000 ...
        ...
    CommonBuffers Small: Total 32 Available 8 Used 24 TotalBytes 16384
        [ 1] dt _TRACKING_DATA 0xfffffa80059a2798 VA 0xfffffa8005aeb000 LA 0x1168eb000 ...
        [ 2] dt _TRACKING_DATA 0xfffffa80059a27e8 VA 0xfffffa8005aeb200 LA 0x1168eb200 ...
        ...
```

See also

USB 3.0 Extensions

!xhci_dumpall

Universal Serial Bus (USB) Drivers

!usb3kd.xhci_trb

12/5/2018 • 2 minutes to read • Edit Online

The **!usb3kd.xhci_trb** extension displays one or more transfer request blocks (TRBs) used by a USB 3.0 host controller

```
!usb3kd.xhci_trb VirtualAddress Count
!usb3kd.xhci_trb PhysicalAddress Count 1
```

Parameters

VirtualAddress
Virtual address of a TRB.

PhysicalAddress
Physical address of a TRB.

Count
The number of consecutive TRBs to display, starting at *VirtualAddress* or *PhysicalAddress*.

1
Specifies that the address is a physical address.

DLL

Usb3kd.dll

Remarks

The output the **!xhci_trb** command is based on the data structures maintained by the USB 3.0 host controller driver (UsbXhci.sys). For more information about the USB 3.0 host controller driver and other drivers in the USB stack, see USB Driver Stack Architecture.

Examples

In the following example, **0x844d7c00** is the virtual address of a TRB. The **1** is the count, which specifies how many consecutive TRBs to display.

```
0: kd> !xhci_trb 0x844d7c00 1

        [  0] ISOCH        0x844d7c00 CycleBit 1 IOC 0 CH 1 BEI 0 InterrupterTarget 1 TransferLength  2688
TDSize  0 TBC 0 TLBPC 2 Frame 0x3D2
```

In the following example, **0x0dced7c00** is the physical address of a TRB. The **4** is the count, which specifies how many consecutive TRBs to display. The **1** specifies that the address is a physical address.

```
0: kd> !xhci_trb 0x0dced7c00 4 1

        [  0] ISOCH        0xdced7c00 CycleBit 1 IOC 0 CH 1 BEI 0 InterrupterTarget 1 TransferLength  2688
TDSize  0 TBC 0 TLBPC 2 Frame 0x3D2
        [  1] EVENT_DATA   0xdced7c10 CycleBit 1 IOC 1 CH 0 BEI 1 InterrupterTarget 1 Data 0x194c9bcf001b0001
PacketId 27 Frame 0x194c9bcf TotalBytes 2688
        [  2] ISOCH        0xdced7c20 CycleBit 1 IOC 0 CH 1 BEI 0 InterrupterTarget 1 TransferLength  1352
TDSize  2 TBC 0 TLBPC 2 Frame 0x3D2
        [  3] NORMAL       0xdced7c30 CycleBit 1 IOC 0 CH 1 BEI 0 InterrupterTarget 1 TransferLength  1336
TDSize  0
```

See also

USB 3.0 Extensions

!xhci_dumpall

Universal Serial Bus (USB) Drivers

!usb3kd.xhci_transferring

12/5/2018 • 2 minutes to read • Edit Online

The **!usb3kd.xhci_transferring** extension displays a transfer ring (used by a USB 3.0 host controller) until it detects a cycle bit change.

```
!usb3kd.xhci_transferring VirtualAddress
!usb3kd.xhci_transferring PhysicalAddress 1
```

Parameters

VirtualAddress
Virtual address of the transfer ring.

PhysicalAddress
Physical address of the transfer ring.

1
Specifies that the address is a physical address.

DLL

Usb3kd.dll

Remarks

The output of the **!xhci_transferring** command is based on the data structures maintained by the USB 3.0 host controller driver (UsbXhci.sys). For more information about the USB 3.0 host controller driver and other drivers in the USB stack, see USB Driver Stack Architecture.

The transfer ring is a structure used by the USB 3.0 host controller driver to maintain a list of transfer request blocks (TRBs). This command takes the virtual or physical address of a transfer ring, but displays the physical address of the TRBs. This is done so the command can correctly traverse LINK TRBs.

Examples

To obtain the address of the transfer ring, look at the output of the **!xhci_deviceslots** command. In the following example, the virtual address of the transfer ring is 0xfffffa8005b2fe00.

```
3: kd> !usb3kd.xhci_deviceslots 0xfffffa800523a2d0

## Dumping dt _DEVICESLOT_DATA 0xfffffa80051a3300
-----------------------------------------------
DeviceContextBase: VA 0xfffffa8005a41000 LA 0x116841000 !wdfcommonbuffer 0x57ffa6ff9b8 Size 4096

## [1] SlotID : dt USBXHCI!_USBDEVICE_DATA 0xfffffa800598c7d0 dt _SLOT_CONTEXT32 0xfffffa8005a42000
-------------------------------------------------------------------------------------------------
    USB\VID_125F&PID_312A ADATA Technology Co., Ltd.
    SlotEnabled IsDevice NumberOfTTs 0 TTThinkTime 0
    Speed: Super PortPathDepth: 1 PortPath: [ 2 ] DeviceAddress: 1
    !device_info_from_pdo 0xfffffa800597d720
    DeviceContextBuffer: VA 0xfffffa8005a42000 LA 0x116842000 !wdfcommonbuffer 0x57ffa7009b8 Size 4096
    InputDeviceContextBuffer: VA 0xfffffa8005b2d000 LA 0x11692d000 !wdfcommonbuffer 0x57ffa674958 Size 4096
    ...

    [3] DeviceContextIndex : dt USBXHCI!_ENDPOINT_DATA 0xfffffa8005b394e0 dt _ENDPOINT_CONTEXT32
0xfffffa8005a42060 ES_RUNNING
    -------------------------------------------------------------------------------------------------------------
----
       ...
            CurrentRingBufferData: VA 0xfffffa8005b2fe00 LA 0x11692fe00 !wdfcommonbuffer 0x57ffa67c988 Size
512
            Current:  !xhci_transferring 0xfffffa8005b2fe00
            PendingTransferList:
                [0] dt _TRANSFER_DATA 0xfffffa8005b961b0 !urb 0xfffffa8005b52be8 !wdfrequest 0x57ffa469fd8
TransferState_Pending
```

Now you can pass the address of the transfer ring to the **!xhci_transferring** command.

```
kd> !xhci_transferring 0xfffffa8005b2fe00

        [  0] NORMAL       0x000000011692fe00 CycleBit 1 IOC 0 BEI 0 InterrupterTarget 0 TransferLength    13
TDSize  0
        [  1] EVENT_DATA   0x000000011692fe10 CycleBit 1 IOC 1 BEI 0 InterrupterTarget 0 Data  0
0xfffffa8005986850 TotalBytes 13
        [  2] NORMAL       0x000000011692fe20 CycleBit 1 IOC 0 BEI 0 InterrupterTarget 0 TransferLength    13
TDSize  0
        [  3] EVENT_DATA   0x000000011692fe30 CycleBit 1 IOC 1 BEI 0 InterrupterTarget 0 Data  0
0xfffffa8005b96210 TotalBytes 13
        [  4] NORMAL       0x000000011692fe40 CycleBit 1 IOC 0 BEI 0 InterrupterTarget 0 TransferLength    13
TDSize  0
        [  5] EVENT_DATA   0x000000011692fe50 CycleBit 1 IOC 1 BEI 0 InterrupterTarget 0 Data  0
0xfffffa8005b96210 TotalBytes 13
```

See also

USB 3.0 Extensions

!xhci_dumpall

Universal Serial Bus (USB) Drivers

USB 2.0 Extensions

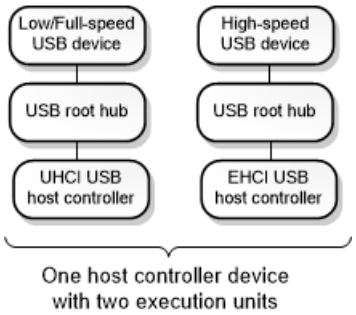
12/5/2018 • 2 minutes to read • Edit Online

This section describes the USB 2.0 debugger extension commands. These commands display information from data structures maintained by drivers in the USB 2.0 driver stack. For more information about these three drivers, see USB Driver Stack Architecture.

The USB 2.0 debugger extension commands are implemented in Usbkd.dll. To load the Usbkd commands, enter **.load usbkd.dll** in the debugger.

USB 2.0 Tree

The USB 2.0 tree contains the device nodes that represent execution units on EHCI host controller devices along with the child nodes that represent hubs and connected devices. This diagram shows an example of a USB 2.0 tree.



The diagram shows one physical host controller device that has two execution units. Each execution unit appears as a device node in the Plug and Play device tree. One execution unit appears as a UHCI USB host controller node, and the other execution unit shows as an EHCI USB host controller node. Each of those nodes has a child node that represents a USB root hub. Each root hub has a single child node that represents a connected USB device.

Notice that the diagram is not a tree in the sense that not all nodes descend from a single parent node. However, when we use the term *USB 2.0 tree*, we are referring to the set of device nodes that represent execution units on EHCI host controller devices along with the nodes for hubs and connected devices.

Getting started with USB 2.0 debugging

To start debugging a USB 2.0 issue, enter the **!usb2tree** command. The **!usb2tree** command displays a list of commands and addresses that you can use to investigate host controllers, hubs, ports, devices, endpoints, and other elements of the USB 2.0 tree.

In this section

- **!usbkd.usbhelp**
- **!usbkd._ehcidd**
- **!usbkd._ehciep**
- **!usbkd._ehciframe**
- **!usbkd._ehciqh**
- **!usbkd._ehciregs**
- **!usbkd._ehcisitd**

- **!usbkd._ehcistq**
- **!usbkd._ehcitd**
- **!usbkd._ehcitfer**
- **!usbkd._ehciitd**
- **!usbkd.doesdumphaveusbdata**
- **!usbkd.isthisdumpasyncissue**
- **!usbkd.urbfunc**
- **!usbkd.usb2**
- **!usbkd.usb2tree**
- **!usbkd.usbchain**
- **!usbkd.usbdevobj**
- **!usbkd.usbdpc**
- **!usbkd.ehci_info_from_fdo**
- **!usbkd.usbdevh**
- **!usbkd.usbep**
- **!usbkd.usbfaildata**
- **!usbkd.usbhcdext**
- **!usbkd.usbdstatus**
- **!usbkd.usbhcdhccontext**
- **!usbkd.usbhcdlist**
- **!usbkd.usbhcdlistlogs**
- **!usbkd.usbhcdlog**
- **!usbkd.usbhcdlogex**
- **!usbkd.usbhcdpnp**
- **!usbkd.usbhcdpow**
- **!usbkd.hub2_info_from_fdo**
- **!usbkd.usbhuberr**
- **!usbkd.usbhubext**
- **!usbkd.usbhubinfo**
- **!usbkd.usbhublog**
- **!usbkd.usbhubmddevext**
- **!usbkd.usbhubmdpd**
- **!usbkd.usbhubpd**
- **!usbkd.usbhubs**
- **!usbkd.usblist**
- **!usbkd.usbpo**
- **!usbkd.usbpdos**
- **!usbkd.usbpdoxls**
- **!usbkd.usbpnp**
- **!usbkd.usbportisasyncadv**
- **!usbkd.usbportmdportlog**
- **!usbkd.usbportmddcontext**
- **!usbkd.usbportmddevext**
- **!usbkd.usbtriage**
- **!usbkd.usbtt**
- **!usbkd.usbtx**

- **!usbkd.usbusb2ep**
- **!usbkd.usbusb2tt**
- **!usbkd.usbver**

Related topics

USB 3.0 Extensions

RCDRKD Extensions

!usbkd.usbhelp

12/5/2018 • 2 minutes to read • Edit Online

The **!usbkd.usbhelp** command displays help for the USB 2.0 debugger extension commands.

DLL

Usbkd.dll

See also

USB 2.0 Debugger Extensions

Universal Serial Bus (USB) Drivers

!usbkd._ehcidd

12/5/2018 • 2 minutes to read • Edit Online

The **!usbkd._ehcidd** command displays information from a **usbehci!_DEVICE_DATA** structure.

```
!usbkd._ehcidd StructAddr
```

Parameters

StructAddr
Address of a **usbehci!_DEVICE_DATA** structure. To find addresses of **usbehci!_DEVICE_DATA** structures, use **!usbhcdext** or **!usbhcdlist**.

DLL

Usbkd.dll

Examples

Here is one way to get the address of a **usbehci!_DEVICE_DATA** structure. First enter **!usbkd.usbhcdlist**.

```
0: kd> !usbkd.usbhcdlist

MINIPORT List @ fffff80001e5bbd0

## List of EHCI controllers

!drvobj ffffe00001fd33a0 dt USBPORT!_USBPORT_MINIPORT_DRIVER ffffe00001f48bd0 Registration Packet
ffffe00001f48c08

01. Xxxx Corporation PCI: VendorID Xxxx DeviceID Xxxx RevisionId 0002
    !devobj ffffe0000781a050
    !ehci_info ffffe0000781a1a0
    Operational Registers ffffd00021fb8420
    Device Data ffffe0000781bda0
    ...
```

In the preceding output, `ffffe0000781bda0` is the address of a **_DEVICE_DATA** structure.

Now pass the structure address to **!_ehcidd**

```
0: kd> !usbkd._ehcidd ffffe0000781bda0

*USBEHCI DEVICE DATA ffffe0000781bda0
** dt usbehci!_DEVICE_DATA ffffe0000781bda0

get_field_ulong ffffe0000781bda0 usbehci!_DEVICE_DATA Flags
*All Enpoints list:
head @ ffffe0000781bdb0 f_link ffffe0000781bdb0 b_link ffffe0000781bdb0
AsyncQueueHead ffffd00021cf5000 !_ehciqh ffffd00021cf5000
    PhysicalAddress: 0xde79a000
    NextQh: ffffd00021cf5000 Hlink de79a002
    PrevQh: ffffd00021cf5000
```

See also

USB 2.0 Debugger Extensions

Universal Serial Bus (USB) Drivers

!usbkd._ehciep

6/25/2019 • 2 minutes to read • Edit Online

The **!usbkd._ehciep** command displays information from a **usbehci!_ENDPOINT_DATA** structure. Use this command to display information about asynchronous endpoints (that is, control and bulk endpoints).

```
!usbkd._ehciep StructAddr
```

Parameters

StructAddr
Address of a **usbehci!_ENDPOINT_DATA** structure. To find addresses of **usbehci!_ENDPOINT_DATA** structures, use **!usbhcdext** and **!usblist**.

DLL

Usbkd.dll

Examples

This example shows one way to get the address of a **usbehci!_ENDPOINT_DATA** structure. Start with the **!usb2tree** command.

```
0: kd> !usbkd.usb2tree
...
2)!ehci_info ffffe0000206e1a0 !devobj ffffe0000206e050 PCI: VendorId 8086 DeviceId 293c RevisionId 0002
    RootHub !hub2_info ffffe000024a61a0 !devstack ffffe000024a6050
        Port 1: !port2_info ffffe000026dd000
        Port 2: !port2_info ffffe000026ddb40
        Port 3: !port2_info ffffe000026de680 !devstack ffffe00001ec3060
            !device2_info ffffe00001ec31b0 (USB Mass Storage Device: Xxx Corporation)
        Port 4: !port2_info ffffe000026df1c0
```

In the preceding output, the address of the device extension of the FDO is displayed as the argument of the DML command **!ehci_info ffffe0000206e1a0**. Either click the DML command or pass the address of the device extension to **!usbhcdext**.

```
0: kd> !usbkd.usbhcdext ffffe0000206e1a0
...
DeviceHandleList: !usblist ffffe0000206f3b8, DL
DeviceHandleDeletedList: !usblist ffffe0000206f3c8, DL [Empty]
GlobalEndpointList: !usblist ffffe0000206f388, EP
...
```

The preceding output displays the command **!usblist ffffe0000206f388, EP**. Use this command to display a list of endpoints.

```
0: kd> !usblist ffffe0000206f388, EP
list: ffffe0000206f388 EP
...
dt usbport!_HCD_ENDPOINT ffffe000026dc970  !usbep ffffe000026dc970
common buffer bytes 0x00003000 (12288) @ va 0000000021e6e000 pa 00000000d83c9000
Device Address: 0x01, ep 0x81 Bulk In Flags: 00000041 dt _USB_ENDPOINT_FLAGS ffffe000026dc990
... usbehci!_ENDPOINT_DATA ffffe000026dcc38
...
```

In the preceding output, `ffffe000026dcc38` is the address of a **usbehci!_ENDPOINT_DATA** structure. Pass this address to **!_ehciep**.

```
0: kd> !usbkd._ehciep ffffe000026dcc38
*USBEHCI
dt usbehci!_ENDPOINT_DATA ffffe000026dcc38
Flags: 0x00000000
dt usbehci!_HCD_QUEUEHEAD_DESCRIPTOR ffffd00021e6e080
*HwQH ffffd00021e6e080
HwQH
     HwQH.HLink dea2e002
     HwQH.EpChars 02002101
         DeviceAddress: 0x1
         IBit: 0x0
         EndpointNumber: 0x1
         EndpointSpeed: 0x2 HcEPCHAR_HighSpeed
         DataToggleControl: 0x0
         HeadOfReclimationList: 0x0
         MaximumPacketLength: 0x200 - 512
   ...
current slot 0000000000000000
slot[0] dt usbehci!_ENDPOINT_SLOT ffffe000026dcdb8 - slot_NotBusy
----
     ffffd00021e6e100
     dt usbehci!_HCD_TRANSFER_DESCRIPTOR ffffd00021e6e100
    tdphys: d83c9100'200 txlen 00000000 tx ffffd00000000041 flags 6d4e695d  _BUSY _SLOT_RESET
    Next_qTD: d83c9200'd83c9180 AltNext_qTD 77423c00'41
    NextTD: ffffd00021e6e200 AltNextTD ffffd00021e6e180 SlotNextTd ffffd00021e6e200 tok 00000c00  Xbytes x0
(0)
.
     ffffd00021e6e200
     dt usbehci!_HCD_TRANSFER_DESCRIPTOR ffffd00021e6e200
    tdphys: d83c9200'5000 txlen 00000000 tx ffffd00000000041 flags 6d4e695d  _BUSY _SLOT_RESET
    Next_qTD: d83c9280'd83c9180 AltNext_qTD 77423c00'41
    NextTD: ffffd00021e6e280 AltNextTD ffffd00021e6e180 SlotNextTd ffffd00021e6e280 tok 00000c00  Xbytes x0
(0)
...
```

See also

USB 2.0 Debugger Extensions

Universal Serial Bus (USB) Drivers

!usbkd._ehciframe

12/5/2018 • 2 minutes to read • Edit Online

The **!usbkd._ehciframe** command displays an EHCI miniport FrameListBaseAddress periodic list entry chain indexed by a frame number.

```
!usbkd._ehciframe StructAddr, FrameNumber
```

Parameters

StructAddr
Address of a **usbehci!_DEVICE_DATA** structure.

FrameNumber
Frame number in the range 0 through 1023.

DLL

Usbkd.dll

See also

USB 2.0 Debugger Extensions

Universal Serial Bus (USB) Drivers

!usbkd._ehciqh

12/5/2018 • 2 minutes to read • Edit Online

The **!usbkd._ehciqh** command displays information from a **usbehci!_HCD_QUEUEHEAD_DESCRIPTOR** structure. Use this command to display information about asynchronous endpoints (that is, control and bulk endpoints).

```
!usbkd._ehciqh StructAddr
```

Parameters

StructAddr
Address of a **usbehci!_HCD_QUEUEHEAD_DESCRIPTOR** structure.

DLL

Usbkd.dll

See also

USB 2.0 Debugger Extensions

Universal Serial Bus (USB) Drivers

!usbkd._ehciregs

12/5/2018 • 2 minutes to read • Edit Online

The **!usbkd._ehciregs** command displays the operational and root hub port status registers of a USB EHCI host controller.

```
!usbkd._ehciregs StructAddr[, NumPorts]
```

Parameters

StructAddr
Address of a **usbehci!_HC_OPERATIONAL_REGISTER** structure. To find the address of a **usbehci!_HC_OPERATIONAL_REGISTER** structure, use **!usbkd.usbhcdlist**.

NumPorts
The number of root hub port status registers to display.

DLL

Usbkd.dll

Examples

Here is one way to get the address of a **usbehci!_HC_OPERATIONAL_REGISTER** structure. First enter **!usbkd.usbhcdlist**.

```
0: kd> !usbkd.usbhcdlist
MINIPORT List @ fffff80001e5bbd0

## List of EHCI controllers

!drvobj ffffe00001fd33a0 dt USBPORT!_USBPORT_MINIPORT_DRIVER ...
...
02. Xxxx Corporation PCI: VendorID Xxxx DeviceID Xxxx RevisionId 0002
    !devobj ffffe00001ca1050
    !ehci_info ffffe00001ca11a0
    Operational Registers ffffd000228bf020
```

In the preceding output, `ffffd000228bf020` is the address of a **_HC_OPERATIONAL_REGISTER** structure.

Now pass the structure address to **!_ehciregs**. In this example, the second argument limits the display to two root hub port status registers.

```
0: kd> !usbkd._ehciregs ffffd000228bf020, 2
*(ehci)HC_OPERATIONAL_REGISTER ffffd000228bf020
    USBCMD 00010001
    .HostControllerRun: 1
    .HostControllerReset: 0
    .FrameListSize: 0
    .PeriodicScheduleEnable: 0
    .AsyncScheduleEnable: 0
    .IntOnAsyncAdvanceDoorbell: 0
    .HostControllerLightReset: 0
    .InterruptThreshold: 1
```

```
.InterruptThreshold: 1
.ParkModeEnable: 0
.ParkModeCount: 0

USBSTS 00002008
.UsbInterrupt: 0
.UsbError: 0
.PortChangeDetect: 0
.FrameListRollover: 1
.HostSystemError: 0
.IntOnAsyncAdvance: 0
----
.HcHalted: 0
.Reclimation: 1
.PeriodicScheduleStatus: 0
.AsyncScheduleStatus: 0

USBINTR 0000003f
.UsbInterrupt: 1
.UsbError: 1
.PortChangeDetect: 1
.FrameListRollover: 1
.HostSystemError: 1
.IntOnAsyncAdvance: 1
PeriodicListBase dec8e000
AsyncListAddr dec91000
PortSC[0] 00001000
    PortConnect x0
    PortConnectChange x0
    PortEnable x0
    PortEnableChange x0
    OvercurrentActive x0
    OvercurrentChange x0
    ForcePortResume x0
    PortSuspend x0
    PortReset x0
    HighSpeedDevice x0
    LineStatus x0
    PortPower x1
    PortOwnedByCC x0
    PortIndicator x0
    PortTestControl x0
    WakeOnConnect x0
    WakeOnDisconnect x0
    WakeOnOvercurrent x0
PortSC[1] 00001000
    PortConnect x0
    PortConnectChange x0
    PortEnable x0
    PortEnableChange x0
    OvercurrentActive x0
    OvercurrentChange x0
    ForcePortResume x0
    PortSuspend x0
    PortReset x0
    HighSpeedDevice x0
    LineStatus x0
    PortPower x1
    PortOwnedByCC x0
    PortIndicator x0
    PortTestControl x0
    WakeOnConnect x0
    WakeOnDisconnect x0
```

See also

USB 2.0 Debugger Extensions

!usbkd._ehcisitd

12/5/2018 • 2 minutes to read • Edit Online

The **!usbkd._ehcisitd** command displays information from a **usbehci!_HCD_SI_TRANSFER_DESCRIPTOR**

```
!usbkd._ehcisitd StructAddr
```

Parameters

StructAddr
Address of a **usbehci!_HCD_SI_TRANSFER_DESCRIPTOR** structure.

DLL

Usbkd.dll

See also

USB 2.0 Debugger Extensions

Universal Serial Bus (USB) Drivers

!usbkd._ehcistq

12/5/2018 • 2 minutes to read • Edit Online

The **!usbkd._ehcistq** command displays a **usbehci!_HCD_QUEUEHEAD_DESCRIPTOR** structure.

```
!usbkd._ehciep StructAddr
```

Parameters

StructAddr
Address of a **usbehci!_HCD_QUEUEHEAD_DESCRIPTOR** structure.

DLL

Usbkd.dll

See also

USB 2.0 Debugger Extensions

Universal Serial Bus (USB) Drivers

!usbkd._ehcitd

12/5/2018 • 2 minutes to read • Edit Online

The **!usbkd._ehcitd** command displays information from a **usbehci!_TRANSFER_CONTEXT** structure. Use this command to display information about asynchronous endpoints (that is, control and bulk endpoints).

```
!usbkd._ehcitd StructAddr
```

Parameters

StructAddr
Address of a **usbehci!_TRANSFER_CONTEXT** structure.

DLL

Usbkd.dll

Examples

This example shows one way to get the address of a **usbehci!_TRANSFER_CONTEXT** structure. Use **!_ehciep** to display information about an endpoint.

```
0: kd> !_ehciep ffffe000001ab618
*USBEHCI
dt usbehci!_ENDPOINT_DATA ffffe000001ab618
Flags: 0x00000000
dt usbehci!_HCD_QUEUEHEAD_DESCRIPTOR ffffd00021e65080
*HwQH ffffd00021e65080
HwQH
     HwQH.HLink dea2e002
     HwQH.EpChars 02002201
         DeviceAddress: 0x1
         IBit: 0x0
         EndpointNumber: 0x2
    ...
slot[0] dt usbehci!_ENDPOINT_SLOT ffffe000001ab798 - slot_NotBusy
----
     ffffd00021e65100
     dt usbehci!_HCD_TRANSFER_DESCRIPTOR ffffd00021e65100
    ....
```

In the preceding output, `ffffd00021e65100` is the address of a **usbehci!_TRANSFER_CONTEXT** structure. Pass this address to **!_ehcitd**.

```
0: kd> !_ehcitd ffffd00021e65100
*USBEHCI TD 21e65100
Sig 20td
     qTD
     Next_qTD: d83cc200
     AltNext_qTD: d83cc180
     Token: 0x00000c00
         PingState: 0x0
         SplitXstate: 0x0
         MissedMicroFrame: 0x0
         XactErr: 0x0
         BabbleDetected: 0x0
         DataBufferError: 0x0
         Halted: 0x0
         Active: 0x0
         Pid: 0x0 - HcTOK_Out
         ErrorCounter: 0x3
         C_Page: 0x0
         InterruptOnComplete: 0x0
         BytesToTransfer: 0x0
         DataToggle: 0x0
     BufferPage[0]: 0x 0bad0-000  0bad0000  BufferPage64[0]: 00000000
     BufferPage[1]: 0x 0bad0-000  0bad0000  BufferPage64[1]: 00000000
     BufferPage[2]: 0x 0bad0-000  0bad0000  BufferPage64[2]: 00000000
     BufferPage[3]: 0x 0bad0-000  0bad0000  BufferPage64[3]: 00000000
     BufferPage[4]: 0x 0bad0-000  0bad0000  BufferPage64[4]: 00000000
Packet:00 52 e6 21 00 d0 ff ff
PhysicalAddress: d83cc100
EndpointData: 001ab618
TransferLength : 0000001f
TransferContext: 00000000
Flags: 00000041
    TD_FLAG_BUSY
NextHcdTD: 21e65200
AltNextHcdTD: 21e65180
SlotNextHcdTD: 21e65200
```

See also

USB 2.0 Debugger Extensions

Universal Serial Bus (USB) Drivers

!usbkd._ehcitfer

12/5/2018 • 2 minutes to read • Edit Online

The **!usbkd._ehcitfer** command displays information from a **usbehci!_HCD_TRANSFER_DESCRIPTOR** structure.

```
!usbkd._ehcitfer StructAddr
```

Parameters

StructAddr
Address of a **usbehci!_HCD_TRANSFER_DESCRIPTOR** structure.

DLL

Usbkd.dll

See also

USB 2.0 Debugger Extensions

Universal Serial Bus (USB) Drivers

!usbkd._ehciitd

12/5/2018 • 2 minutes to read • Edit Online

The **!usbkd._ehciitd** command displays information from a **usbehci!_HCD_HSISO_TRANSFER_DESCRIPTOR** structure.

```
!usbkd._ehciitd StructAddr
```

Parameters

StructAddr
Address of a **usbehci!_HCD_HSISO_TRANSFER_DESCRIPTOR** structure.

DLL

Usbkd.dll

See also

USB 2.0 Debugger Extensions

Universal Serial Bus (USB) Drivers

!usbkd.doesdumphaveusbdata

12/5/2018 • 2 minutes to read • Edit Online

The **!usbkd.doesdumphaveusbdata** command checks to see which types of USB data are in a crash dump file that was generated as a result of **Bug Check 0xFE**.

```
!usbkd.doesdumphaveusbdata
```

DLL

Usbkd.dll

Remarks

Use this command only when you are debugging a crash dump file that was generated as a result of **Bug Check 0xFE: BUGCODE_USB_DRIVER**.

Examples

Here is an example of the output of **!doesdumphaveusbdata**

```
1: kd> !analyze -v
*** ...
BUGCODE_USB_DRIVER (fe)
...
1: kd> !usbkd.doesdumphaveusbdata

Retrieving crashdump information Please Wait...

Checking for GuidUsbHubPortArrayData information...
There is no data for this GUID in the mini dump.
No data to print

Checking for GuidUsbHubExt information...
There is no data for this GUID in the mini dump.
No data to print

Checking for GuidUsbPortLog information...
GuidUsbPortLog Exists with PORT Log Size = 8000

Checking for GuidUsbPortContextData information...
GuidUsbPortContextData Exists with Data Length = 4c8

Checking for GuidUsbPortExt information...
GuidUsbPortExt Exists (DEVICE_EXTENSION + DeviceDataSize ) = 2250
```

See also

USB 2.0 Debugger Extensions

Universal Serial Bus (USB) Drivers

!usbkd.isthisdumpasyncissue

12/5/2018 • 2 minutes to read • Edit Online

The **!usbkd.isthisdumpasyncissue** command checks a crash dump file, generated by **Bug Check 0xFE**, to see whether the likely cause of the crash was an Interrupt on Async Advance issue associated with a USB EHCI host controller.

```
!usbkd.isthisdumpasyncissue
```

DLL

Usbkd.dll

Remarks

Use this command only when you are debugging a crash dump file that was generated as a result of **Bug Check 0xFE: BUGCODE_USB_DRIVER**.

Examples

Here is an example of the output of **!usbkd.isthisdumpasyncissue**.

```
1: kd> !analyze -v
*** ...
BUGCODE_USB_DRIVER (fe)
...
1: kd> !usbkd.isthisdumpasyncissue
This is *NOT* Async on Advance Issue because the EndPointData is NULL
```

See also

USB 2.0 Debugger Extensions

Universal Serial Bus (USB) Drivers

!usbkd.urbfunc

12/5/2018 • 2 minutes to read • Edit Online

The **!usbkd.urbfunc** command displays the name of a URB function code.

```
!usbkd.urbfunc FunctionCode
```

Parameters

FunctionCode
The hexadecimal value of a URB function code. These codes are defined in usb.h.

DLL

Usbkd.dll

Examples

Here is an example of the output of **!urbfunc**.

```
0: kd> !usbkd.urbfunc 0xA

URB_FUNCTION_ISOCH_TRANSFER (0xA)
```

See also

USB 2.0 Debugger Extensions

Universal Serial Bus (USB) Drivers

!usbkd.usb2

12/5/2018 • 2 minutes to read • Edit Online

The **!usbkd.usb2** command displays a list of USB endpoints that have USB 2.0 scheduling information.

```
!usbkd.usb2 DeviceExtension
```

Parameters

DeviceExtension
Address of the device extension for the functional device object (FDO) of a USB host controller.

DLL

Usbkd.dll

Examples

Here is one way to find the address of the device extension for the FDO of a USB host controller. First enter **!usbkd.usb2tree**.

```
0: kd> !usbkd.usb2tree

EHCI MINIPORT(s) dt usbport!_USBPORT_MINIPORT_DRIVER ffffe00001f48bd0
...
2)!ehci_info ffffe00001ca11a0 !devobj ffffe00001ca1050 PCI: VendorId 8086 DeviceId 293c RevisionId 0002
...
```

In the preceding output, the address of the device extension of the FDO is displayed as the argument of the DML command **!ehci_info ffffe00001ca11a0**. Pass the address of the device extension to the **!usb2** command.

```
0: kd> !usbkd.usb2 ffffe00001ca11a0

Sig: HFDO
Hcd FDO Extension:
----------
----------
dt usbport!_HCD_ENDPOINT ffffe0000212d970  !usbep ffffe0000212d970
    Tt 0000000000000000 Device Address: 0x00, ep 0x81 Interrupt In
    dt _USB2LIB_ENDPOINT_CONTEXT ffffe000023b60f0    dt _USB2_EP ffffe000023b6100
    Period,offset,Ordinal(32,0,0)   smask,cmask(00,00  ........ , ........) maxpkt 1
```

See also

USB 2.0 Debugger Extensions

Universal Serial Bus (USB) Drivers

!usbkd.usb2tree

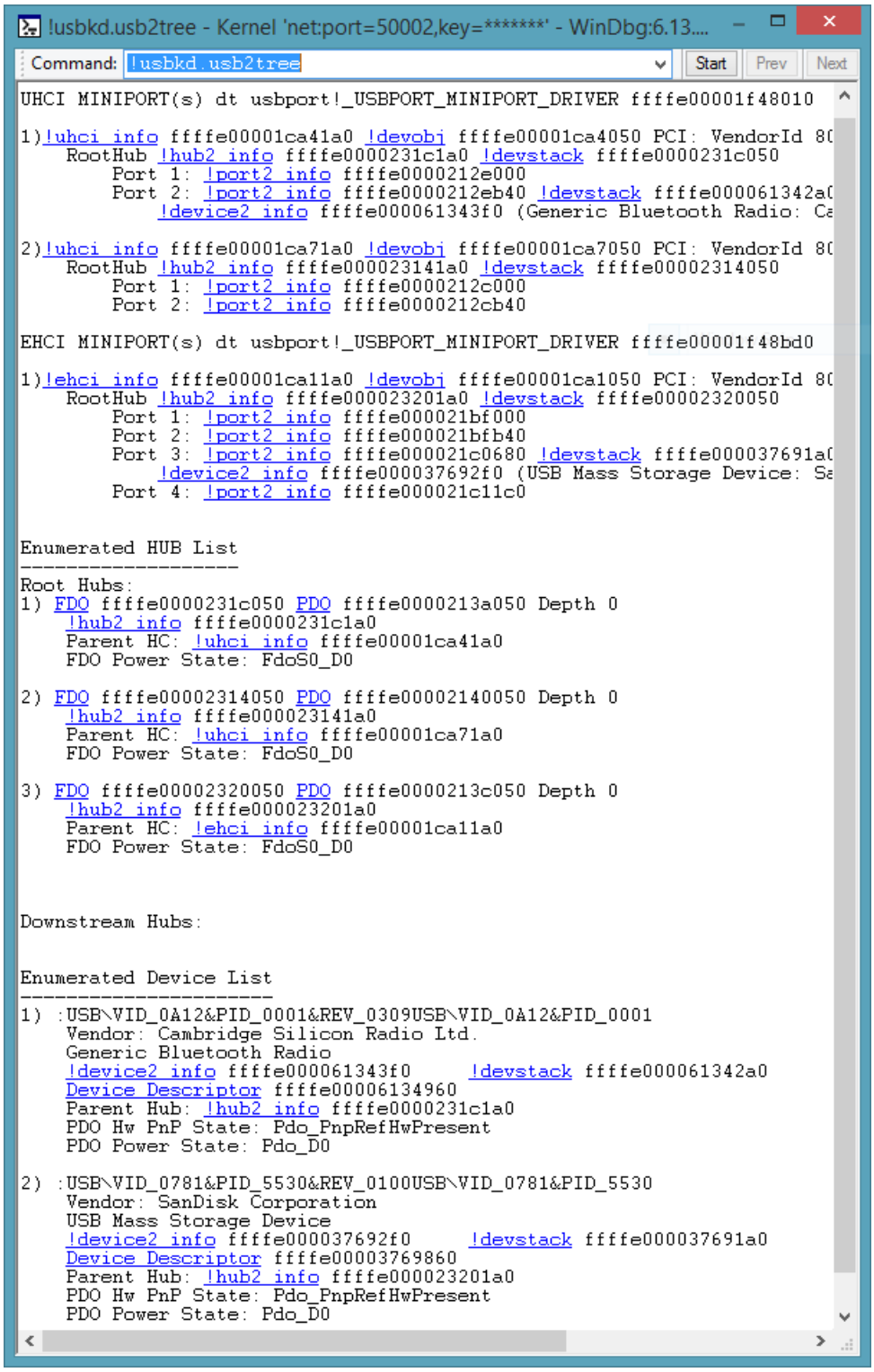
12/5/2018 • 2 minutes to read • Edit Online

The **!usbkd.usb2tree** command displays USB 2.0 tree.

```
!usbkd.usb2tree
```

Examples

This screen shot shows and example of the output of the **!usb2tree** command.



The output shows one EHCI execution unit and two UHCI execution units. The execution units shown in this example happen to be on a single USB host controller device. The output also shows the root hubs and connected devices.

The output uses Using Debugger Markup Language (DML) to provide links. The links execute commands that give detailed information related to objects in the tree. For example, you could click one of the **!devobj** links to get information about the functional device object associated with the EHCI execution unit. As an alternative to clicking the link, you could enter the command manually: **!devobj ffffe00001ca7050**

Note The DML feature is available in WinDbg, but not in Visual Studio or KD.

DLL

Usb3kd.dll

Remarks

The **!usb2tree** command is the parent command for many of the USB 2.0 debugger extensions commands. The information displayed by these commands is based on data structures maintained by these drivers:

- usbehci.sys (miniport driver for USB 2 host controller)
- usbuhci.sys (miniport driver for USB 2 host controller)
- usbport.sys (port driver for USB 2 host controller)
- usbhub.sys (USB 2 hub driver)

For more information about these drivers, see USB Driver Stack Architecture.

See also

USB 2.0 Debugger Extensions

Universal Serial Bus (USB) Drivers

!usbkd.usbchain

12/5/2018 • 2 minutes to read • Edit Online

The **!usbkd.usbchain** command displays a USB device chain starting at a specified PDO, and going back to the root hub.

```
!usbkd.usbchain PDO
```

Parameters

PDO
Address of the physical device object (PDO) of a device that is connected to a USB hub.

DLL

Usbkd.dll

Examples

Here is one way to find the address of the PDO of a USB device. First enter **!usbkd.usb2tree**.

```
 kd> !usbkd.usb2tree
...
2)!ehci_info ffffe00001ca11a0 !devobj ffffe00001ca1050 PCI: VendorId 8086 DeviceId 293c RevisionId 0002
    RootHub !hub2_info ffffe000023201a0 !devstack ffffe00002320050
        Port 1: !port2_info ffffe000021bf000
        Port 2: !port2_info ffffe000021bfb40
        Port 3: !port2_info ffffe000021c0680 !devstack ffffe00007c882a0
...
```

In the preceding output, the address of the PDO is the argument of the suggested command **!devstack ffffe00007c882a0**. Pass the address of the PDO to **!usbkd.usbchain**.

```
0: kd> !usbkd.usbchain ffffe00007c882a0

usbchain
****************************************************************************
HUB PDO ffffe00007c882a0 on port 3 !usbhubext ffffe00007c883f0 ArmedForWake = 0
VID Xxxx PID Xxxx REV 0100  Xxxx Corporation
    HUB #3 FDO ffffe00002320050 , !usbhubext ffffe000023201a0  HWC_ARM=0
    ROOT HUB PDO(ext) @ffffe0000213c1a0
        ROOT HUB FDO @ffffe00001ca1050, !usbhcdext ffffe00001ca11a0 PCI Vendor:Device:...
```

See also

USB 2.0 Debugger Extensions

Universal Serial Bus (USB) Drivers

!usbkd.usbdevobj

12/5/2018 • 2 minutes to read • Edit Online

The **!usbkd.usbdevobj** command displays information from a USB device object.

```
!usbkd.usbdevobj DeviceObject
```

Parameters

DeviceObject
Address of a **_DEVICE_OBJECT** structure.

DLL

Usbkd.dll

See also

USB 2.0 Debugger Extensions

Universal Serial Bus (USB) Drivers

!usbkd.usbdpc

6/25/2019 • 2 minutes to read • Edit Online

The **!usbkd.usbdpc** command displays information stored in an **_XDPC_CONTEXT** structure.

```
!usbkd.usbdpc StructAddr
```

Parameters

StructAddr
Address of a **usbport!_XDPC_CONTEXT** structure. To get the XDPC list for a USB host controller, use the **!usbkd.usbhcdext** command.

DLL

Usbkd.dll

Examples

Here is one way to find the address of a **usbport!_XDPC_CONTEXT** structure. First enter **!usbkd.usb2tree**.

```
0: kd> !usbkd.usb2tree
...
UHCI MINIPORT(s) dt usbport!_USBPORT_MINIPORT_DRIVER ffffe00001e77010
...
4)!uhci_info ffffe00001c7d1a0 !devobj ffffe00001c7d050 PCI: VendorId...
...
```

In the preceding output, the address of the device extension of the FDO is displayed as the argument of the DML command **!uhci_info ffffe00001c7d1a0**.

Either click the DML command or pass the address of the device extension to **!usbhcdext** to get the XDPC list.

```
0: kd> !usbkd.usbhcdext ffffe00001c7d1a0
...
## XDPC List

01) dt USBPORT!_XDPC_CONTEXT ffffe00001c7df18
02) dt USBPORT!_XDPC_CONTEXT ffffe00001c7db88
03) dt USBPORT!_XDPC_CONTEXT ffffe00001c7dd50
04) dt USBPORT!_XDPC_CONTEXT ffffe00001c7e0e0
...
```

In the preceding output, `ffffe00001c7df18` is the address of an **_XDPC_CONTEXT** structure. Pass this address to **!usbdpc**.

```
0: kd> !usbkd.usbdpc ffffe00001c7df18

dt USBPORT!_XDPC_CONTEXT ffffe00001c7df18

## XDPC HISTORY (latest at boottom)

##      EVENT                STATE                   NEXT

[01] Ev_Xdpc_End          XDPC_Running            XDPC_Enabled
[02] Ev_Xdpc_Signal       XDPC_Enabled            XDPC_DpcQueued
[03] Ev_Xdpc_Signal       XDPC_DpcQueued          XDPC_DpcQueued
[04] Ev_Xdpc_Worker       XDPC_DpcQueued          XDPC_Running
[05] Ev_Xdpc_Signal       XDPC_Running            XDPC_Signaled
[06] Ev_Xdpc_End          XDPC_Signaled           XDPC_DpcQueued
[07] Ev_Xdpc_Worker       XDPC_DpcQueued          XDPC_Running
[08] Ev_Xdpc_End          XDPC_Running            XDPC_Enabled
```

See also

USB 2.0 Debugger Extensions

Universal Serial Bus (USB) Drivers

!usbkd.ehci_info_from_fdo

6/25/2019 • 2 minutes to read • Edit Online

The **!usbkd.ehci_info_from_fdo** command displays information about a USB host controller.

```
!usbkd.ehci_info_from_fdo fdo
```

Parameters

fdo
Address of the functional device object (FDO) of a UHCI or EHCI USB host controller. You can get the address of the FDO from the output of the **!usb2tree** command.

DLL

Usbkd.dll

Examples

First use the **!usb2tree** command to get the address of the FDO.

```
0: kd> !usbkd.usb2tree

EHCI MINIPORT(s) dt usbport!_USBPORT_MINIPORT_DRIVER ffffe00001f48bd0

1)!ehci_info ffffe00001ca11a0 !devobj ffffe00001ca1050 PCI: VendorId 8086 DeviceId 293c RevisionId 0002
...
```

In the preceding output, you can see that the address of the FDO of the USB host controller is `ffffe00001ca1050`.
Pass the address of the FDO to **!ehci_info_from_fdo**.

```
0: kd> !usbkd.ehci_info_from_fdo ffffe00001ca1050

HC Flavor 1000  FDO ffffe00001ca1050
Root Hub: FDO ffffe00002320050 !hub2_info ffffe000023201a0
Operational Registers ffffd000228bf020
Device Data ffffe00001ca2da0
dt USBPORT!_FDO_EXTENSION ffffe00001ca15a0
DM Timer Flags ffffe00001ca16d4
FDO Flags ffffe00001ca16d0
HCD Log ffffe00001ca11a0

DeviceHandleList: !usblist ffffe00001ca23b8, DL
DeviceHandleDeletedList: !usblist ffffe00001ca23c8, DL [Empty]
GlobalEndpointList: !usblist ffffe00001ca2388, EP
EpNeoStateChangeList: !usblist ffffe00001ca2370, SC [Empty]
GlobalTtListHead: !usblist ffffe00001ca23a8, TT [Empty]
BusContextHead: !usblist ffffe00001ca16b0, BC

## Pending Requests

[001] dt USBPORT!_USB_IOREQUEST_CONTEXT ffffe00001ca1450 Tag: AddD Obj: ffffe00001ca11a0
...

## XDPC List

01) dt USBPORT!_XDPC_CONTEXT ffffe00001ca1f18
...

## PnP FUNC HISTORY (latest at bottom)

[01] IRP_MN_QUERY_CAPABILITIES
...
```

See also

USB 2.0 Debugger Extensions

Universal Serial Bus (USB) Drivers

!usbkd.usbdevh

6/25/2019 • 2 minutes to read • Edit Online

The **!usbkd.usbdevh** command displays information about a USB device handle.

```
!usbkd.usbdevh StructAddr
```

Parameters

StructAddr
Address of a **usbport!_USBD_DEVICE_HANDLE** structure. To get the device handle list for a USB host controller, use the **!usbkd.usbhcdext** command.

DLL

Usbkd.dll

Examples

Here is one way to find the address of a **usbport!_USBD_DEVICE_HANDLE** structure. First enter **!usbkd.usb2tree**.

```
0: kd> !usbkd.usb2tree
...
2)!ehci_info ffffe00001ca11a0 !devobj ffffe00001ca1050 PCI: VendorId 8086 DeviceId 293c RevisionId 0002
     ...
```

In the preceding output, the address of the device extension of the FDO is displayed as the argument of the DML command **!ehci_info ffffe00001ca11a0**.

Either click the DML command or pass the address of the device extension to **!usbhcdext** to get the device handle list.

```
0: kd> !usbkd.usbhcdext ffffe00001ca11a0

HC Flavor 1000  FDO ffffe00001ca1050
Root Hub: FDO ffffe00002320050 !hub2_info ffffe000023201a0
Operational Registers ffffd000228bf020
Device Data ffffe00001ca2da0
dt USBPORT!_FDO_EXTENSION ffffe00001ca15a0
DM Timer Flags ffffe00001ca16d4
FDO Flags ffffe00001ca16d0
HCD Log ffffe00001ca11a0

DeviceHandleList: !usblist ffffe00001ca23b8, DL
DeviceHandleDeletedList: !usblist ffffe00001ca23c8, DL [Empty]
...
```

Now use the **!usbkd.usblist** command to get the addresses of **usbport!_USBD_DEVICE_HANDLE** structures.

```
0: kd> !usblist ffffe00001ca23b8, DL
list: ffffe00001ca23b8 DL
----------
!usbdevh ffffe000020f9590
SSP [IdleReady] (0)
...
```

In the preceding output, `ffffe000020f9590` is the address of a **_USBD_DEVICE_HANDLE** structure. Pass this address to **!usbdevh**.

```
0: kd> !usbkd.usbdevh ffffe000020f9590

dt USBPORT!_USBD_DEVICE_HANDLE ffffe000020f9590
SSP [IdleReady] (0)
PCI\VEN_8086&DEV_293C  Intel Corporation
Root Hub
DriverName :

## DEVICE HANDLE HISTORY (latest at boottom)

##      EVENT                        STATE                        NEXT

[01] Ev_CreateRoothub_Success     Devh_Created                Devh_Valid

## Referene List: Head(ffffe000020f9668)

[00] dt USBPORT!_DEVH_REF_OBJ ffffe000021944a0  Object: ffffe000020f9590 Tag: dvh+ PendingFlag(0)
[01] dt USBPORT!_DEVH_REF_OBJ ffffe000020bbcb0  Object: ffffe000020ba7e0 Tag: bsCT PendingFlag(0)
[02] dt USBPORT!_DEVH_REF_OBJ ffffe000032b91a0  Object: ffffe0000269e670 Tag: UrbT PendingFlag(1)

## TtList: Head(ffffe000020f9658)


## PipeHandleList: Head(ffffe000020f9640)

[00] dt USBPORT!_USBD_PIPE_HANDLE_I ffffe000020f95e0 !usbep ffffe000020f6970
        Device Address: 0x00, Endpoint Address 0x00 Endpoint Type: Control
[01] dt USBPORT!_USBD_PIPE_HANDLE_I ffffe000023bd278 !usbep ffffe0000212d970
        Device Address: 0x00, Endpoint Address 0x81 Endpoint Type: Interrupt In

Config Information: dt USBPORT!_USBD_CONFIG_HANDLE ffffe000023cd0b0

## Interface List:

[00] dt USBPORT!_USBD_INTERFACE_HANDLE_I ffffe000023bd250
```

See also

USB 2.0 Debugger Extensions

Universal Serial Bus (USB) Drivers

!usbkd.usbep

6/25/2019 • 2 minutes to read • Edit Online

The **!usbkd.usbep** command displays information about a USB endpoint.

```
!usbkd.usbep StructAddr
```

Parameters

StructAddr
Address of a **usbport!_HCD_ENDPOINT** structure. To get the endpoint list for a USB host controller, use the **!usbkd.usbhcdext** command.

DLL

Usbkd.dll

Examples

Here is one way to find the address of a **usbport!_HCD_ENDPOINT** structure. First enter **!usbkd.usb2tree**.

```
0: kd> !usbkd.usb2tree
...
2)!ehci_info ffffe00001ca11a0 !devobj ffffe00001ca1050 PCI: VendorId 8086 DeviceId 293c RevisionId 0002
     ...
```

In the preceding output, the address of the device extension of the FDO is displayed as the argument of the DML command **!ehci_info ffffe00001ca11a0**.

Either click the DML command or pass the address of the device extension to **!usbhcdext** to get the global endpoint list.

```
0: kd> !usbkd.usbhcdext ffffe00001ca11a0
...
DeviceHandleList: !usblist ffffe00001ca23b8, DL
DeviceHandleDeletedList: !usblist ffffe00001ca23c8, DL [Empty]
GlobalEndpointList: !usblist ffffe00001ca2388, EP
...
```

Now use the **!usbkd.usblist** command to get the addresses of **_HCD_ENDPOINT** structures.

```
0: kd> !usblist ffffe00001ca2388, EP

list: ffffe00001ca2388 EP
----------
dt usbport!_HCD_ENDPOINT ffffe000020f6970  !usbep ffffe000020f6970
Device Address: 0x00, ep 0x00 Control  Flags: 00000002 dt _USB_ENDPOINT_FLAGS ffffe000020f6990
dt usbport!_ENDPOINT_PARAMETERS ffffe000020f6b18    RootHub Endpoint
...
```

In the preceding output, `ffffe000020f6970` is the address of an **_HCD_ENDPOINT** structure. Pass this address to

!usbkd.usbep.

```
0: kd> !usbep ffffe000020f6970
Device Address: 0x00, Endpoint Address 0x00 Endpoint Type: Control
dt USBPORT!_HCD_ENDPOINT ffffe000020f6970
dt USBPORT!_ENDPOINT_PARAMETERS ffffe000020f6b18
RootHub Endpoint

## Transfer(s) List: (HwPendingListHead)

    [EMPTY]

## Endpoint Reference List: (EpRefListHead)

[00] dt USBPORT!_USBOBJ_REF ffffe000021a64a0 Object ffffe000020f6970 Tag:EPop Endpoint:ffffe000020f6970
[01] dt USBPORT!_USBOBJ_REF ffffe0000021264a0 Object ffffe000020f95e0 Tag:EPpi Endpoint:ffffe000020f6970

## GEP HISTORY (latest at bottom)

##      EVENT                   STATE                   NEXT                    HwEpState

[01] Ev_gEp_Open            GEp_Init                GEp_Paused              ENDPOINT_PAUSE
[02] Ev_gEp_ReqActive       GEp_Paused              GEp_Active              ENDPOINT_ACTIVE
```

See also

USB 2.0 Debugger Extensions

Universal Serial Bus (USB) Drivers

!usbkd.usbfaildata

12/5/2018 • 2 minutes to read • Edit Online

The **!usbkd.usbfaildata** command displays the failure data (if any) stored for a USB device.

```
!usbkd.usbfaildata PDO
```

Parameters

PDO
Address of the physical device object (PDO) of a device that is connected to a USB hub.

DLL

Usbkd.dll

Examples

Here is one way to find the address of the PDO of a USB device. Enter **!usbkd.usb2tree**.

```
 kd> !usbkd.usb2tree
...
2)!ehci_info ffffe00001ca11a0 !devobj ffffe00001ca1050 PCI: VendorId 8086 DeviceId 293c RevisionId 0002
    RootHub !hub2_info ffffe000023201a0 !devstack ffffe00002320050
        Port 1: !port2_info ffffe000021bf000
        Port 2: !port2_info ffffe000021bfb40
        Port 3: !port2_info ffffe000021c0680 !devstack ffffe00007c882a0
...
```

In the preceding output, the address of the PDO appears as the argument of the suggested command **!devstack ffffe00007c882a0**.

Now pass the address of the PDO to **!usbkd.usbfaildata**.

See also

USB 2.0 Debugger Extensions

Universal Serial Bus (USB) Drivers

!usbkd.usbhcdext

6/25/2019 • 2 minutes to read • Edit Online

The **!usbkd.usbhcdext** command displays information from the device extension of a USB host controller or a USB root hub.

```
!usbkd.usbhcdext DeviceExtension
```

Parameters

DeviceExtension
Address of one of the following:

- The device extension for the functional device object (FDO) of a USB host controller.
- The device extension for the physical device object (PDO) a USB root hub.

DLL

Usbkd.dll

Examples

Here is one way to find the address of the device extension for the FDO of an EHCI host controller. First enter **!usbkd.usb2tree**.

```
0: kd> !usbkd.usb2tree

EHCI MINIPORT(s) dt usbport!_USBPORT_MINIPORT_DRIVER ffffe00001f48bd0

1)!ehci_info ffffe00001ca11a0 !devobj ffffe00001ca1050 PCI: VendorId 8086 DeviceId 293c RevisionId 0002
 ...
```

In the preceding output, the address of the device extension of the FDO is displayed as the argument of the DML command **!ehci_info ffffe00001ca11a0**.

Now pass the address of the device extension to the **!usbhcdext** command.

```
0: kd> !usbkd.usbhcdext ffffe00001ca11a0

HC Flavor 1000  FDO ffffe00001ca1050
Root Hub: FDO ffffe00002320050 !hub2_info ffffe000023201a0
Operational Registers ffffd000228bf020
Device Data ffffe00001ca2da0
dt USBPORT!_FDO_EXTENSION ffffe00001ca15a0
DM Timer Flags ffffe00001ca16d4
FDO Flags ffffe00001ca16d0
HCD Log ffffe00001ca11a0

DeviceHandleList: !usblist ffffe00001ca23b8, DL
DeviceHandleDeletedList: !usblist ffffe00001ca23c8, DL [Empty]
GlobalEndpointList: !usblist ffffe00001ca2388, EP
EpNeoStateChangeList: !usblist ffffe00001ca2370, SC [Empty]
GlobalTtListHead: !usblist ffffe00001ca23a8, TT [Empty]
BusContextHead: !usblist ffffe00001ca16b0, BC

## Pending Requests

[001] dt USBPORT!_USB_IOREQUEST_CONTEXT ffffe00001ca1450 Tag: AddD Obj: ffffe00001ca11a0
...

## XDPC List

01) dt USBPORT!_XDPC_CONTEXT ffffe00001ca1f18
...
```

Here is one way to find the address of the device extension for the PDO of a root hub. First enter **!usbkd.usb2tree**.

```
0: kd> !usbkd.usb2tree

EHCI MINIPORT(s) dt usbport!_USBPORT_MINIPORT_DRIVER ffffe00001f48bd0

1)!ehci_info ffffe00001ca11a0 !devobj ffffe00001ca1050 PCI: VendorId 8086 DeviceId 293c RevisionId 0002
    RootHub !hub2_info ffffe000023201a0 !devstack ffffe00002320050
 ...
```

In the preceding output, you can see the address of the FDO of the root hub displayed as the argument to the command **!devstack ffffe00002320050**. Use the **!devstack** command to find the address of the PDO and the PDO device extension.

```
0: kd> !kdexts.devstack ffffe00002320050
  !DevObj           !DrvObj            !DevExt           ObjectName
> ffffe00002320050  \Driver\usbhub     ffffe000023201a0  0000002d
  ffffe0000213c050  \Driver\usbehci    ffffe0000213c1a0  USBPDO-3
...
```

In the preceding output, you can see that the address of the device extension for the PDO of the root hub is `ffffe0000213c1a0`.

Now pass the address of the device extension to the **!usbhcdext** command.

```
0: kd> !usbkd.usbhcdext ffffe0000213c1a0

Root Hub PDO Extension
Parent HC: FDO ffffe00001ca1050 !ehci_info ffffe00001ca11a0
HUB FDO ffffe00002320050 !hub2_info ffffe000023201a0
dt USBPORT!_PDO_EXTENSION ffffe0000213c5a0

## Pending Requests

[001] dt USBPORT!_USB_IOREQUEST_CONTEXT ffffe0000213c450 Tag: RHcr Obj: ffffe0000213c1a0
[002] dt USBPORT!_USB_IOREQUEST_CONTEXT ffffe00003ce5800 Tag: iIRP Obj: ffffe00002182210

## POWER FUNC HISTORY (latest at bottom)

[00] IRP_MN_WAIT_WAKE (PowerSystemHibernate)
...

## PnP STATE LOG (latest at bottom)

##      EVENT                          STATE              NEXT

[01] EvPDO_IRP_MN_START_DEVICE      PnpNotStarted       PnpStarted
```

See also

USB 2.0 Debugger Extensions

Universal Serial Bus (USB) Drivers

!usbkd.usbdstatus

12/5/2018 • 2 minutes to read • Edit Online

The **!usbkd.usbdstatus** command displays the name of a USBD status code.

```
!usbkd.usbdstatus StatusCode
```

Parameters

StatusCode
The hexadecimal value of a USBD status code. These codes are defined in usb.h.

DLL

Usbkd.dll

Examples

Here is an example of the output of **!usbdstatus**.

```
1: kd> !usbkd.usbdstatus 0xC0000008

USBD_STATUS_DATA_OVERRUN (0xC0000008)
```

See also

USB 2.0 Debugger Extensions

Universal Serial Bus (USB) Drivers

!usbkd.usbhcdhccontext

12/5/2018 • 2 minutes to read • Edit Online

The **!usbkd.usbhcdhccontext** command displays the **USB2LIB_HC_CONTEXT** for a USB host controller.

```
!usbkd.usbhcdhccontext DeviceExtension
```

Parameters

DeviceExtension
Address of the device extension for the functional device object (FDO) of a USB host controller.

DLL

Usbkd.dll

See also

USB 2.0 Debugger Extensions

Universal Serial Bus (USB) Drivers

!usbkd.usbhcdlist

6/25/2019 • 2 minutes to read • Edit Online

The **!usbkd.usbhcdlist** command displays information about all USB host controllers that are represented by the USB port driver (Usbport.sys). For information about the USB port driver and the associated miniport drivers, see USB Driver Stack Architecture.

```
!usbkd.usbhcdlist
```

DLL

Usbkd.dll

Examples

Here is an example of a portion of the output of **!usbhcdlist**.

```
0: kd> !usbkd.usbhcdlist
MINIPORT List @ fffff80001e5bbd0

## List of UHCI controllers

!drvobj ffffe00002000060 dt USBPORT!_USBPORT_MINIPORT_DRIVER ffffe00001f48010 Registration Packet
ffffe00001f48048

01
...

## List of EHCI controllers

!drvobj ffffe00001fd33a0 dt USBPORT!_USBPORT_MINIPORT_DRIVER ffffe00001f48bd0 Registration Packet
ffffe00001f48c08

01. Xxxxx Corporation PCI: VendorID Xxxx DeviceID Xxxx RevisionId 0002
    !devobj ffffe00001ca1050
    !ehci_info ffffe00001ca11a0
    Operational Registers ffffd000228bf020
    Device Data ffffe00001ca2da0
    !usbhcdlog ffffe00001ca11a0
    nt!_KINTERRUPT ffffe000020abe78
    Device Capabilities ffffe00001ca135c
    Pending IRP's: 0, Transfers: 0 (Periodic(0), Async(0))
```

See also

USB 2.0 Debugger Extensions

Universal Serial Bus (USB) Drivers

!usbkd.usbhcdlistlogs

12/5/2018 • 2 minutes to read • Edit Online

The **!usbkd.usbhcdlistlogs** command displays a list of all functional device objects (FDOs) associated with the USB port driver (Usbport.sys). The command also displays the complete debug logs for all EHCI Host Controllers.

```
!usbkd.usbhcdlistlogs
```

DLL

Usbkd.dll

Examples

This example shows a portion of the output of the **!usbhcdlistlogs** command.

```
0: kd> !usbkd.usbhcdlistlogs

MINIPORT List @ fffff80001e5bbd0
*flink, blink ffffe00001f48018,ffffe00001f48bd8

[0] entry @: ffffe00001f48010 dt usbport!_USBPORT_MINIPORT_DRIVER ffffe00001f48010
UHCI MINIPORT @ ffffe00001f48010 !drvobj ffffe00002000060 regpkt ffffe00001f48048
02. !devobj ffffe00001ca4050 !usbhcdext ffffe00001ca41a0  HFDO 8086 2938  RHPDO ffffe0000213a050
03. !devobj ffffe00001ca7050 !usbhcdext ffffe00001ca71a0  HFDO 8086 2937  RHPDO ffffe00002140050

[1] entry @: ffffe00001f48bd0 dt usbport!_USBPORT_MINIPORT_DRIVER ffffe00001f48bd0
EHCI MINIPORT @ ffffe00001f48bd0 !drvobj ffffe00001fd33a0 regpkt ffffe00001f48c08
01. !devobj ffffe00001ca1050 !usbhcdext ffffe00001ca11a0  HFDO 8086 293c  RHPDO ffffe0000213c050

LOG@: ffffe00001ca11b8
>LOG mask = 3ff idx = fff69e8d (28d)
*LOG: ffffe000020191a0  LOGSTART: ffffe00002014000 *LOGEND: ffffe0000201bfe0 # -1
[ 000] ffffe000020191a0 Tmt0 0000000000000000 0000000000000000 0000000000000000
[ 001] ffffe000020191c0 _Pop 0000000000000003 0000000000003000 0000000000000000
...
[ 005] ffffe00002019240 chgZ 0000000000000000 0000000000b7228f 0000000000000000
[ 006] ffffe00002019260 nes- 0000000000000000 0000000000000000 0000000000000000
[ 007] ffffe00002019280 nes+ ffffe00001ca2370 ffffe00001ca2370 00000000000ced39
...
[1014] ffffe00002019060 tmo4 0000000000000000 0000000000000040 ffffe00001ca1c20
...
[1022] ffffe00002019160 xdw2 ffffe00001ca1b88 ffffe00001ca1050 0000000000000002
[1023] ffffe00002019180 xdB0 ffffe00001ca1b88 ffffe00001ca1050 0000000000000000
```

The command output shows two FDOs that represent UHCI host controlers and one FDO that represents an EHCI host controller. Then the output shows the debug log for the one EHCI host controller.

See also

USB 2.0 Debugger Extensions

Universal Serial Bus (USB) Drivers

!usbkd.usbhcdlog

6/25/2019 • 2 minutes to read • Edit Online

The **!usbkd.usbhcdlog** command displays a portion of the debug log for a USB host controller.

```
!usbkd.usbhcdlog DeviceExtension[, NumberOfEntries]
```

Parameters

DeviceExtension
Address of the device extension for the functional device object (FDO) of a UHCI or EHCI USB host controller.

NumberOfEntries
The number of log entries to display. To display the entire log, set this parameter to -1.

DLL

Usbkd.dll

Examples

Here is one way to find the address of the device extension for the FDO of a USB host controller. First enter **!usbkd.usb2tree**.

```
0 kd> !usbkd.usb2tree

EHCI MINIPORT(s) dt usbport!_USBPORT_MINIPORT_DRIVER ffffe00001f48bd0
...

2)!ehci_info ffffe00001ca11a0 !devobj ffffe00001ca1050 PCI: VendorId 8086 DeviceId 293c RevisionId 0002
...
```

In the preceding output, the address of the device extension of the FDO is displayed as the argument of the DML command **!ehci_info ffffe00001ca11a0**.

Now pass the address of the device extension to the **!usbhcdlog** command. In this example, the second argument limits the display to four log entries.

```
0: kd> !usbkd.usbhcdlog ffffe00001ca11a0, 4

LOG@: ffffe00001ca11b8
>LOG mask = 3ff idx = fff68e95 (295)
*LOG: ffffe0000201 92a0  LOGSTART: ffffe00002014000 *LOGEND: ffffe0000201bfe0 # 4
[ 000] ffffe0000201 92a0 xSt0 ffffe00001ca1b88 0000000000000006 0000000000000001
[ 001] ffffe0000201 92c0 xnd8 ffffe00001ca1b88 ffffe00001ca1050 0000000000000000
[ 002] ffffe0000201 92e0 xnd0 ffffe00001ca1b88 ffffe00001ca1050 0000000000000000
[ 003] ffffe00002019300 gNX0 0000000000000000 0000000000000000 ffffe00001ca1b88
```

See also

USB 2.0 Debugger Extensions

!usbkd.usbhcdlogex

12/5/2018 • 2 minutes to read • Edit Online

The **!usbkd.usbhcdlogex** command displays an annotated debug log for a USB host controller.

```
!usbkd.usbhcdlogex DeviceExtension[, NumberOfEntries]
```

Parameters

DeviceExtension
Address of the device extension for the functional device object (FDO) of a UHCI or EHCI USB host controller.

NumberOfEntries
The number of log entries to display. To display the entire log, set this parameter to -1.

DLL

Usbkd.dll

Examples

Here is one way to find the address of the device extension for the FDO of a USB host controller. First enter **!usbkd.usb2tree**.

```
0 kd> !usbkd.usb2tree

EHCI MINIPORT(s) dt usbport!_USBPORT_MINIPORT_DRIVER ffffe00001f48bd0
...

2)!ehci_info ffffe00001ca11a0 !devobj ffffe00001ca1050 PCI: VendorId 8086 DeviceId 293c RevisionId 0002
...
```

In the preceding output, the address of the device extension of the FDO is displayed as the argument of the DML command **!ehci_info ffffe00001ca11a0**.

Now pass the address of the device extension to the **!usbhcdlogex** command. In this example, the second argument limits the display to 20 log entries.

```
0: kd> !usbkd.usbhcdlogex ffffe00001ca11a0, 20
LOG@: ffffe00001ca11b8
>LOG mask = 3ff idx = fff68e95 (295)
*LOG: ffffe0000020192a0  LOGSTART: ffffe00002014000 *LOGEND: ffffe0000201bfe0 # 20
[ 000] ffffe0000020192a0 xSt0 ffffe00001ca1b88 0000000000000006 0000000000000001
[ 001] ffffe0000020192c0 xnd8 ffffe00001ca1b88 ffffe00001ca1050 0000000000000000
[ 002] ffffe0000020192e0 xnd0 ffffe00001ca1b88 ffffe00001ca1050 0000000000000000
//
// USBPORT_Xdpc_End()    - USBPORT_Core_UsbHcIntDpc_Worker() DPC/XDPC: 2 of 4

[ 003] ffffe00002019300 gNX0 0000000000000000 0000000000000000 ffffe00001ca1b88
[ 004] ffffe00002019320 xbg1 ffffe00001ca1b88 ffffe00001ca1050 0000000000000000
[ 005] ffffe00002019340 xbg0 ffffe00001ca1b88 ffffe00001ca1050 ffffe00001ca22e8
//
// USBPORT_Xdpc_iBegin() - USBPORT_Core_UsbHcIntDpc_Worker() DPC/XDPC: 2 of 4

[ 006] ffffe00002019360 tmo4 0000000000000000 0000000000000040 ffffe00001ca1c20
[ 007] ffffe00002019380 tmo3 0000000000000000 0000000000989680 ffffe00001ca1c20
[ 008] ffffe0000020193a0 tmo2 0000000000000000 00000000000003e8 ffffe00001ca1c20
[ 009] ffffe0000020193c0 tmo1 0000000000000000 000000000002625a ffffe00001ca1c20
[ 010] ffffe0000020193e0 tmo0 00000000000003e8 ffffe00001ca1b88 ffffe00001ca1c20
[ 011] ffffe00002019400 hci0 0000000000000000 0000000000000000 0000000000000000
[ 012] ffffe00002019420 xSt0 ffffe00001ca1b88 0000000000000008 0000000000000003
[ 013] ffffe00002019440 xdw4 ffffe00001ca1b88 0000000000000000 0000000000000000
[ 014] ffffe00002019460 xdw2 ffffe00001ca1b88 ffffe00001ca1050 0000000000000002
[ 015] ffffe00002019480 xdB0 ffffe00001ca1b88 ffffe00001ca1050 0000000000000000
//
// USBPORT_Xdpc_Worker_HcIntDpc()  DPC/XDPC: 2 of 4

[ 016] ffffe0000020194a0 iDP- 0000000000000000 0000000000b73e26 0000000000000000
//
// USBPORT_IsrDpc() - Exit()

[ 017] ffffe0000020194c0 xSt0 ffffe00001ca1b88 0000000000000007 0000000000000002
[ 018] ffffe0000020194e0 Xsi1 ffffe00001ca1b88 0000000000000000 0000000000000000
//
// USBPORT_Xdpc_iSignal()- USBPORT_Core_UsbHcIntDpc_Worker() DPC/XDPC: 2 of 4

[ 019] ffffe00002019500 chgZ 0000000000000000 0000000000b73e26 0000000000000000
```

See also

USB 2.0 Debugger Extensions

Universal Serial Bus (USB) Drivers

!usbkd.usbhcdpnp

12/5/2018 • 2 minutes to read • Edit Online

The **!usbkd.usbhcdpnp** command displays the Plug and Play (PnP) state history for a USB host controller or root hub.

```
!usbkd.usbhcdpnp DeviceExtension
```

Parameters

DeviceExtension
Address of one of the following:

- The device extension for the functional device object (FDO) of a USB host controller.
- The device extension for the physical device object (PDO) a USB root hub.

DLL

Usbkd.dll

Examples

Here is one way to find the address of the device extension for the FDO of USB host controller. First enter **!usbkd.usb2tree**.

```
0: kd> !usbkd.usb2tree

UHCI MINIPORT(s) dt usbport!_USBPORT_MINIPORT_DRIVER ffffe0000090c3d0
...
4)!uhci_info ffffe00001c8f1a0 !devobj ffffe00001c8f050 PCI: VendorId 8086 DeviceId 2938 RevisionId 0002
...
```

In the preceding output, the address of the device extension of the FDO is displayed as the argument of the DML command **!uhci_info ffffe00001c8f1a0**.

Now pass the address of the device extension to the **!usbhcdpnp** command.

```
0: kd> !usbkd.usbhcdpnp ffffe00001c8f1a0

## PNP STATE LOG (latest at bottom)

##      EVENT                          STATE              NEXT

[01] EvFDO_IRP_MN_START_DEVICE      PnpNotStarted       PnpStarted
[02] EvFDO_IRP_MN_QBR_RH            PnpStarted          PnpStarted
```

Here is one way to find the address of the device extension for the PDO of a root hub. First enter **!usbkd.usb2tree**.

```
4)!uhci_info ffffe00001c8f1a0 !devobj ffffe00001c8f050 PCI: VendorId 8086 DeviceId 2938 RevisionId 0002
    RootHub !hub2_info ffffe00000d941a0 !devstack ffffe00000d94050
```

In the preceding output, you can see the address of the FDO of the root hub displayed as the argument to the command **!devstack ffffe00000d94050**. Use the **!devstack** command to find the address of the PDO and the PDO device extension.

```
0: kd> !kdexts.devstack ffffe00000d94050
  !DevObj           !DrvObj            !DevExt           ObjectName
> ffffe00000d94050  \Driver\usbhub     ffffe00000d941a0  0000006b
  ffffe00000ed4050  \Driver\usbuhci    ffffe00000ed41a0  USBPDO-2
```

In the preceding output, you can see that the address of the device extension for the PDO of the root hub is `ffffe00000ed41a0`.

Now pass the address of the device extension to the **!usbhcdpnp** command.

```
0: kd> !usbkd.usbhcdpnp ffffe00000ed41a0

## PNP STATE LOG (latest at bottom)

##      EVENT                          STATE              NEXT

[01] EvPDO_IRP_MN_START_DEVICE      PnpNotStarted       PnpStarted
```

See also

USB 2.0 Debugger Extensions

Universal Serial Bus (USB) Drivers

!usbkd.usbhcdpow

12/5/2018 • 2 minutes to read • Edit Online

The **!usbkd.usbhcdpow** command displays the power state history for a USB host controller or root hub.

```
!usbkd.usbhcdpow DeviceExtension
```

Parameters

DeviceExtension
Address of one of the following:

- The device extension for the functional device object (FDO) of a USB host controller.
- The device extension for the physical device object (PDO) a USB root hub.

DLL

Usbkd.dll

Examples

Here is one way to find the address of the device extension for the FDO of an EHCI host controller. First enter **!usbkd.usb2tree**.

```
0: kd> !usbkd.usb2tree
...

2)!ehci_info ffffe00001ca11a0 !devobj ffffe00001ca1050 PCI: VendorId 8086 DeviceId 293c RevisionId 0002
     ...
```

In the preceding output, the address of the device extension of the FDO is displayed as the argument of the DML command **!ehci_info ffffe00001ca11a0**.

Now pass the address of the device extension to the **!usbhcdpow** command.

```
0: kd> !usbkd.usbhcdpow ffffe00001ca11a0

dt USBPORT!_FDO_EXTENSION ffffe00001ca15a0

## State History (latest at bottom)

##      EVENT                              STATE                              NEXT

[00] FdoPwrEv_D0_DoSetD0_2              FdoPwr_D0_WaitWorker2              FdoPwr_D0_WaitSyncUsb2
dt:0 ms
[01] FdoPwrEv_SyncUsb2_DoChirp          FdoPwr_D0_WaitSyncUsb2             FdoPwr_D0_WaitSyncUsb2
dt:0 ms
[02] FdoPwrEv_Rh_SetPowerSys            FdoPwr_D0_WaitSyncUsb2             FdoPwr_D0_WaitSyncUsb2
dt:0 ms
[03] FdoPwrEv_Rh_SetD0                  FdoPwr_D0_WaitSyncUsb2             FdoPwr_D0_WaitSyncUsb2
dt:0 ms
[04] FdoPwrEv_SyncUsb2_Complete         FdoPwr_D0_WaitSyncUsb2             FdoPwr_WaitSx
dt:50 ms
[05] FdoPwrEv_Rh_Wake                   FdoPwr_WaitSx                      FdoPwr_WaitSx
dt:3412 ms
[06] FdoPwrEv_Rh_Wake                   FdoPwr_WaitSx                      FdoPwr_WaitSx
dt:283872 ms
[07] FdoPwrEv_Rh_Wake                   FdoPwr_WaitSx                      FdoPwr_WaitSx
dt:25481267 ms
```

Here is one way to find the address of the device extension for the PDO of a root hub. First enter **!usbkd.usb2tree**.

```
0: kd> !usbkd.usb2tree
...

2)!ehci_info ffffe00001ca11a0 !devobj ffffe00001ca1050 PCI: VendorId 8086 DeviceId 293c RevisionId 0002
    RootHub !hub2_info ffffe000023201a0 !devstack ffffe00002320050
        ...
```

In the preceding output, you can see the address of the FDO of the root hub displayed as the argument to the command **!devstack ffffe00002320050**. Use the **!devstack** command to find the address of the PDO and the PDO device extension.

```
0: kd> !kdexts.devstack ffffe00002320050
  !DevObj           !DrvObj            !DevExt           ObjectName
> ffffe00002320050  \Driver\usbhub     ffffe000023201a0  0000002d
  ffffe0000213c050  \Driver\usbehci    ffffe0000213c1a0  USBPDO-3
...
```

In the preceding output, you can see that the address of the device extension for the PDO of the root hub is `ffffe0000213c1a0`.

Now pass the address of the device extension to the **!usbhcdpow** command.

```
0: kd> !usbkd.usbhcdpow ffffe0000213c1a0

dt USBPORT!_FDO_EXTENSION ffffe0000213c5a0

## State History (latest at bottom)

##      EVENT                              STATE                              NEXT

...
```

See also

USB 2.0 Debugger Extensions

Universal Serial Bus (USB) Drivers

!usbkd.hub2_info_from_fdo

12/5/2018 • 2 minutes to read • Edit Online

The **!usbkd.hub2_info_from_fdo** command displays information about a USB hub.

```
!usbkd.hub2_info_from_fdo FDO
```

Parameters

FDO
Address of the functional device object (FDO) for a USB hub.

DLL

Usbkd.dll

Examples

Here is one way to find the address of the FDO for a USB hub. First enter **!usbkd.usb2tree**.

```
0: kd> !usbkd.usb2tree
...
2)!ehci_info ffffe00001ca11a0 !devobj ffffe00001ca1050 PCI: VendorId 8086 DeviceId 293c RevisionId 0002
    RootHub !hub2_info ffffe000023201a0 !devstack ffffe00002320050
```

In the preceding output, the address of the FDO for the hub appears as the argument of the suggested command **!devstack ffffe00002320050**.

Now pass the address of the FDO to the **!hub2_info_from_fdo** command.

```
0: kd> !usbkd.hub2_info_from_fdo ffffe00002320050
usbhubext
**************************************************************************

FDO ffffe00002320050 PDO ffffe0000213c050 HubNumber# 3
dt USBHUB!_DEVICE_EXTENSION_HUB ffffe000023201a0
!usbhublog ffffe000023201a0
RemoveLock ffffe00002320668
FdoFlags ffffe00002320ba0

CurrentPowerIrp: System (0000000000000000) Device (0000000000000000)

ObjReferenceList: !usblist ffffe00002320b70, RL
ExceptionList: !usblist ffffe00002321498, EL [Empty]
DmTimerListHead: !usblist ffffe00002321040, TL [Empty]
PdoRemovedListHead: !usblist ffffe00002321478, PL [Empty]
PdoPresentListHead: !usblist ffffe00002321468, PL
WorkItemListHead: !usblist ffffe00002320c80, WI [Empty]
SshBusyListHead: !usblist ffffe00002320dc0, BL


## PnP FUNC HISTORY (latest at bottom)

01. IRP_MN_QUERY_DEVICE_RELATIONS
...
```

```
## POWER FUNC HISTORY (latest at bottom)

01. IRP_MN_QUERY_POWER    - PowerSystemHibernate
...

## HARD RESET STATE HISTORY (latest at bottom)

##     EVENT                           STATE                                NEXT

01. HRE_Pause                       HReset_WaitReady                     HReset_Paused
...

## PNP STATE HISTORY (latest at bottom)

##     EVENT                           STATE                                NEXT

01. Ev_SYSTEM_POWER                 FDO_WaitPnpStop                      FDO_WaitPnpStop
...

## POWER STATE HISTORY (latest at bottom)

##     EVENT                           STATE                                NEXT

01. Ev_SET_POWER_S0                 FdoSx_Dx                             FdoWaitS0IoComplete_Dx
...

## BUS STATE HISTORY (latest at bottom)

##     EVENT                           STATE                                NEXT

01. BE_BusSuspend                   BS_BusPause                          BS_BusSuspend
...

SSH_EnabledStatus: [SSH_ENABLED_VIA_POWER_POLICY]

## SSH STATE HISTORY (latest at bottom)

##     EVENT                           STATE                                NEXT

01. SSH_Event_ResumeHubComplete     SSH_State_HubPendingResume           SSH_State_HubActive
...

## PORT DATA

PortData 1: !port2_info ffffe000021bf000 Port State = PS_WAIT_CONNECT PortChangeLock: 0, Pcq_State:
Pcq_Run_Idle
     PDO 0000000000000000
...
```

See also

USB 2.0 Debugger Extensions

Universal Serial Bus (USB) Drivers

!usbkd.usbhuberr

12/5/2018 • 2 minutes to read • Edit Online

The **!usbkd.usbhuberr** command displays a USB hub error record.

```
!usbkd.usbhuberr StructAddr
```

Parameters

StructAddr
Address of a **usbhub!_HUB_EXCEPTION_RECORD** structure.

DLL

Usbkd.dll

Examples

Here is one way to find the address of a **usbhub!_HUB_EXCEPTION_RECORD**. First enter **!usbkd.usb2tree**.

```
0: kd> !usbkd.usb2tree
...
2)!ehci_info ffffe00001ca11a0 !devobj ffffe00001ca1050 PCI: VendorId 8086 DeviceId 293c RevisionId 0002
    RootHub !hub2_info ffffe000023201a0 !devstack ffffe00002320050
      ...
```

In the preceding output, you can see the suggested command **!devstack ffffe00002320050**. Enter this command.

```
0: kd> !kdexts.devstack ffffe000011f7050

  !DevObj           !DrvObj            !DevExt           ObjectName
> ffffe000011f7050  \Driver\usbhub     ffffe000011f71a0  0000006f
  ffffe00000a21050  \Driver\usbehci    ffffe00000a211a0  USBPDO-8
...
```

In the preceding output, `ffffe000011f71a0` is the address of the device extension for the functional device object (FDO) of the hub. Pass the address of the device extension to **!usbkd.usbhubext**.

```
0: kd> !usbkd.usbhubext ffffe000011f71a0

FDO ffffe000011f7050 PDO ffffe00000a21050 HubNumber# 7
dt USBHUB!_DEVICE_EXTENSION_HUB ffffe000011f71a0
!usbhublog ffffe000011f71a0
RemoveLock ffffe000011f7668
FdoFlags ffffe000011f7ba0

CurrentPowerIrp: System (0000000000000000) Device (0000000000000000)

ObjReferenceList: !usblist ffffe000011f7b70, RL
ExceptionList: !usblist ffffe000011f8498, EL [Empty]
...
```

In the preceding output, `ffffe000011f8498` is the address of the exception list. If the exception list is not empty, it will contain addresses of **_HUB_EXCEPTION_RECORD** structures.

See also

USB 2.0 Debugger Extensions

Universal Serial Bus (USB) Drivers

!usbkd.usbhubext

12/5/2018 • 2 minutes to read • Edit Online

The **!usbkd.usbhubext** command displays information about a USB hub.

```
!usbkd.usbhubext DeviceExtension
```

Parameters

DeviceExtension
Address of one of the following:

- The device extension for the functional device object (FDO) of a USB hub.
- The device extension for the physical device object (PDO) of a device that is connected to a USB hub.

DLL

Usbkd.dll

Examples

Here is one way to find the address of the device extension for the FDO of USB hub. First enter **!usbkd.usb2tree**.

```
0: kd> !usbkd.usb2tree
...
2)!ehci_info ffffe00001ca11a0 !devobj ffffe00001ca1050 PCI: VendorId 8086 DeviceId 293c RevisionId 0002
    RootHub !hub2_info ffffe000023201a0 !devstack ffffe00002320050
```

In the preceding output, you can see the suggested command **!devstack ffffe00002320050**. Enter this command.

```
0: kd> !kdexts.devstack ffffe00002320050

  !DevObj           !DrvObj            !DevExt           ObjectName
> ffffe00002320050  \Driver\usbhub     ffffe000023201a0  0000002d
  ffffe0000213c050  \Driver\usbehci    ffffe0000213c1a0  USBPDO-3
...
```

In the preceding output, you can see that the address of the device extension for the FDO of the hub is `ffffe000023201a0`.

Now pass the address of the device extension to the **!usbkd.usbhubext** command.

```
0: kd> !usbkd.usbhubext ffffe000023201a0

FDO ffffe00002320050 PDO ffffe0000213c050 HubNumber# 3
dt USBHUB!_DEVICE_EXTENSION_HUB ffffe000023201a0
!usbhublog ffffe000023201a0
RemoveLock ffffe00002320668
FdoFlags ffffe00002320ba0

CurrentPowerIrp: System (0000000000000000) Device (0000000000000000)

ObjReferenceList: !usblist ffffe00002320b70 -RL
```

```
ObjReferenceList: !usblist ffffe00002320b70, RL
ExceptionList: !usblist ffffe00002321498, EL [Empty]
DmTimerListHead: !usblist ffffe00002321040, TL [Empty]
PdoRemovedListHead: !usblist ffffe00002321478, PL [Empty]
PdoPresentListHead: !usblist ffffe00002321468, PL
WorkItemListHead: !usblist ffffe00002320c80, WI [Empty]
SshBusyListHead: !usblist ffffe00002320dc0, BL


## PnP FUNC HISTORY (latest at bottom)

01. IRP_MN_QUERY_DEVICE_RELATIONS
...
## POWER FUNC HISTORY (latest at bottom)

01. IRP_MN_QUERY_POWER    - PowerSystemHibernate
...

## HARD RESET STATE HISTORY (latest at bottom)

##     EVENT                           STATE                                NEXT

01. HRE_Pause                       HReset_WaitReady                     HReset_Paused
...

## PNP STATE HISTORY (latest at bottom)

##     EVENT                           STATE                                NEXT

01. Ev_SYSTEM_POWER                 FDO_WaitPnpStop                      FDO_WaitPnpStop
...

## POWER STATE HISTORY (latest at bottom)

##     EVENT                           STATE                                NEXT

01. Ev_SET_POWER_S0                 FdoSx_Dx                             FdoWaitS0IoComplete_Dx
...

## BUS STATE HISTORY (latest at bottom)

##     EVENT                           STATE                                NEXT

01. BE_BusSuspend                   BS_BusPause                          BS_BusSuspend
...

SSH_EnabledStatus: [SSH_ENABLED_VIA_POWER_POLICY]

## SSH STATE HISTORY (latest at bottom)

##     EVENT                           STATE                                NEXT

01. SSH_Event_ResumeHubComplete     SSH_State_HubPendingResume           SSH_State_HubActive
...

## PORT DATA

PortData 1: !port2_info ffffe000021bf000 Port State = PS_WAIT_CONNECT PortChangeLock: 0, Pcq_State:
Pcq_Run_Idle
     PDO 0000000000000000
...
```

Here is one way to find the address of the device extension for the PDO of a device that is connected to a USB hub. First enter **!usbkd.usb2tree**.

```
0: kd> !usbkd.usb2tree
...
2)!ehci_info ffffe00001ca11a0 !devobj ffffe00001ca1050 PCI: VendorId 8086 DeviceId 293c RevisionId 0002
    RootHub !hub2_info ffffe000023201a0 !devstack ffffe00002320050
        Port 1: !port2_info ffffe000021bf000
        Port 2: !port2_info ffffe000021bfb40
        Port 3: !port2_info ffffe000021c0680 !devstack ffffe00007c882a0
```

In the preceding output, you can see suggested command **!devstack ffffe00007c882a0**. Enter this command.

```
0: kd> !kdexts.devstack ffffe00007c882a0

  !DevObj           !DrvObj            !DevExt           ObjectName
  ffffe00006ce2260  \Driver\USBSTOR    ffffe00006ce23b0  00000070
> ffffe00007c882a0  \Driver\usbhub     ffffe00007c883f0  USBPDO-4
...
```

In the preceding output, you can see that the address of the device extension for the PDO of the device is `ffffe00007c883f0`.

Now pass the address of the device extension to the **!usbhcdpnp** command.

```
0: kd> !usbkd.usbhubext ffffe00007c883f0

dt USBHUB!_DEVICE_EXTENSION_PDO ffffe00007c883f0
PARENT HUB: FDO ffffe00002320050 !hub2_info ffffe000023201a0
!usbhubinfo ffffe00002320050
PORT NUMBER : 3
IoList: !usblist ffffe00007c888b0, IO
LatchList: !usblist ffffe00007c888e0, LA

## PnP ID's

DeviceId:USB\VID_0781&PID_5530
HardwareId:USB\VID_0781&PID_5530&REV_0100USB\VID_0781&PID_5530
CompatibleId:USB\Class_08&SubClass_06&Prot_50USB\Class_08&SubClass_06USB\Class_08
SerialNumberId:20052444100A47F319CB
UniqueId:3
ProductId:Cruzer

## Pnp Func History (latest at bottom)

01. IRP_MN_QUERY_BUS_INFORMATION
...

## Power Func History (latest at bottom)


## PNP STATE HISTORY (latest at bottom)

##     EVENT                           STATE                                NEXT

01. (6)                           (0)                                PDO_WaitPnpStart
02. Ev_PDO_IRP_MN_START            PDO_WaitPnpStart                    PDO_WaitPnpStop

## POWER STATE HISTORY (latest at bottom)

##     EVENT                           STATE                                NEXT

    [EMPTY]

## HARDWARE STATE HISTORY (latest at bottom)

##     EVENT                           STATE                                NEXT

01. PdoEv_CreatePdo               (0)                               Pdo_Created
02. PdoEv_RegisterPdo             Pdo_Created                        Pdo_HwPresent
03. PdoEv_QBR                     Pdo_HwPresent                      Pdo_PnpRefHwPresent

## IDLE STATE HISTORY (latest at bottom)

##     EVENT                           STATE                                NEXT

    [EMPTY]
```

See also

USB 2.0 Debugger Extensions

Universal Serial Bus (USB) Drivers

!usbkd.usbhubinfo

12/5/2018 • 2 minutes to read • Edit Online

The **!usbkd.hubinfo** command displays information about a USB hub.

```
!usbkd.hubinfo FDO
```

Parameters

FDO
Address of the functional device object (FDO) for a USB hub.

DLL

Usbkd.dll

Examples

Here is one way to find the address of the FDO for a USB hub. First enter **!usbkd.usb2tree**.

```
0: kd> !usbkd.usb2tree
...
2)!ehci_info ffffe00001ca11a0 !devobj ffffe00001ca1050 PCI: VendorId 8086 DeviceId 293c RevisionId 0002
    RootHub !hub2_info ffffe000023201a0 !devstack ffffe00002320050
```

In the preceding output, the address of the FDO for the hub appears as the argument of the suggested command **!devstack ffffe00002320050**.

Now pass the address of the FDO to the **!usbhubinfo** command.

```
0: kd> !usbkd.usbhubinfo ffffe00002320050

    !DevObj ffffe00002320050 !usbhubext ffffe000023201a0
On Host Controller (0x8086, 0x293c)
        Stat_AsyncResumeStartAt: 2437ee39d29bd528
        Stat_AsyncResumeCompleteAt: 24413c77d29bd528
        Stat_AsyncResume: 0x3c(60) ms
        Stat_SyncResumeStartAt: 2437ee39d29bd528
        Stat_SyncResumeCompleteAt: 2437ee39d29bd528
        Stat_SyncResume: 0x0(0) ms
Trap Regs: Event, Port, Event (ffffe000023204d0)
    Enable: 0 Port: 0 Event 00000000
Hub Number: # 3
Number Of Ports: 4
dt usbhub!_USBHUB_FDO_FLAGS ffffe00002320ba0
>Is Root
>Power Switching:
     No Power Switching
>Overcurrent:
     Global Overcurrent
>PortIndicators:
     No PortIndicators present
>AllowWakeOnConnect:
     DO NOT WakeOnConnect
>CURRENT Hub Wake on Connectstate:
     HWC_DISARM:- do not wake system on connect/disconnect event
>CURRENT Bus Wake state:
     BUS_DISARM:- bus not armed for wake by this hub
>CURRENT Wake Detect state (WW Irp):
     HUB_DISARM:- no ww irp pending (HUB_WAKESTATE_DISARMED)
Milliamps/Port : 500ma
Power caps (0 = not reported)
     PortPower_Registry : 0
     PortPower_DeviceStatus : 500
     PortPower_CfgDescriptor : 500
     PortPower_HubStatus : 500
```

See also

USB 2.0 Debugger Extensions

Universal Serial Bus (USB) Drivers

!usbkd.usbhublog

12/5/2018 • 2 minutes to read • Edit Online

The **!usbkd.usbhublog** command displays the debug log for a USB hub.

```
!usbkd.usbhublog DeviceExtension[, NumberOfEntries]
```

Parameters

DeviceExtension
Address of the device extension for the functional device object (FDO) of a USB hub.

NumberOfEntries
The number of log entries to display. To display the entire log, set this parameter to -1.

DLL

Usbkd.dll

Examples

Here is one way to find the address of the device extension for the FDO of a USB hub. First enter **!usbkd.usb2tree**.

```
0: kd> !usbkd.usb2tree
...
2)!ehci_info ffffe00001ca11a0 !devobj ffffe00001ca1050 PCI: VendorId 8086 DeviceId 293c RevisionId 0002
    RootHub !hub2_info ffffe000023201a0 !devstack ffffe00002320050
      ...
```

In the preceding output, you can see the suggested command **!devstack ffffe00002320050**. Enter this command.

```
0: kd> !kdexts.devstack ffffe00002320050

  !DevObj           !DrvObj            !DevExt           ObjectName
> ffffe00002320050  \Driver\usbhub     ffffe000023201a0  0000002d
  ffffe0000213c050  \Driver\usbehci    ffffe0000213c1a0  USBPDO-3
...
```

In the preceding output, `ffffe000023201a0` is the address of the device extension for the FDO of the hub.

Now pass the address of the device extension to **!usbhublog**. In this example, the second argument limits the display to 10 log entries.

```
0: kd> !usbkd.usbhublog ffffe000023201a0, 10

LOG@: ffffe000023201a0 (usbhub!_DEVICE_EXTENSION_HUB)
>LOG mask = ff idx = ffffa333 (33)
*LOG: ffffe00002321ca0  LOGSTART: ffffe00002321640 *LOGEND: ffffe00002323620 # 20
[ 000] ffffe00002321ca0 HDec 0000000000000000 ffffe000002904d0 0000000000000001
[ 001] ffffe00002321cc0 HPCd 0000000000000000 0000000000000002 0000000000000004
[ 002] ffffe00002321ce0 qwk- 0000000000000000 ffffe000021c11c0 0000000000000000
[ 003] ffffe00002321d00 pq-- 0000000000000000 0000000000000002 0000000000000004
[ 004] ffffe00002321d20 _6p4 0000000000000000 0000000000000000 0000000000000004
[ 005] ffffe00002321d40 _6p1 0000000000000000 0000000000000003 0000000000000004
[ 006] ffffe00002321d60 pq++ 0000000000000000 0000000000000003 0000000000000004
[ 007] ffffe00002321d80 pq++ 0000000000000000 0000000000000006 0000000000000004
[ 008] ffffe00002321da0 _6p0 0000000000000000 ffffe000021c11c0 0000000000000004
[ 009] ffffe00002321dc0 pqDP 0000000000000000 ffffe000021c11d8 0000000000000006
```

See also

USB 2.0 Debugger Extensions

Universal Serial Bus (USB) Drivers

!usbkd.usbhubmddevext

12/5/2018 • 2 minutes to read • Edit Online

The **!usbkd.usbhubmddevext** command displays a **usbhub!_DEVICE_EXTENSION_HUB** structure if one is present in a crash dump that was generated as a result of a **Bug Check 0xFE**.

```
!usbkd.usbhubmddevext
```

DLL

Usbkd.dll

Remarks

Use this command only when you are debugging a crash dump file that was generated as a result of **Bug Check 0xFE: BUGCODE_USB_DRIVER**.

See also

USB 2.0 Debugger Extensions

Universal Serial Bus (USB) Drivers

!usbkd.usbhubmdpd

12/5/2018 • 2 minutes to read • Edit Online

The **!usbkd.usbhubmdpd** command displays a **usbhub!_HUB_PORT_DATA** structure if one is present in a crash dump that was generated as a result of **Bug Check 0xFE**.

```
!usbkd.usbhubmdpd [PortNum]
```

Parameters

PortNum
A USB port number. If you specify a port number, this command displays the structure (if one is present) that represents the specified port. If you do not specify a port number, this command displays the structure (if one is present) on which **Bug Check 0xFE** was initiated.

DLL

Usbkd.dll

Remarks

Use this command only when you are debugging a crash dump file that was generated as a result of **Bug Check 0xFE: BUGCODE_USB_DRIVER**.

See also

USB 2.0 Debugger Extensions

Universal Serial Bus (USB) Drivers

!usbkd.usbhubpd

12/5/2018 • 2 minutes to read • Edit Online

The **!usbkd.usbhubpd** command displays information about a USB port.

```
!usbkd.usbhubpd StructAddr
```

Parameters

StructAddr
Address of a **usbhub!_HUB_PORT_DATA** structure. To get the addresses of these structures, use **!usbhubext**.

DLL

Usbkd.dll

Examples

Here is one way to find the address of a **usbhub!_HUB_PORT_DATA**. First enter **!usbkd.usb2tree**.

```
0: kd> !usbkd.usb2tree
...
2)!ehci_info ffffe00001ca11a0 !devobj ffffe00001ca1050 PCI: VendorId 8086 DeviceId 293c RevisionId 0002
    RootHub !hub2_info ffffe000023201a0 !devstack ffffe00002320050
        ...
```

In the preceding output, you can see the suggested command **!devstack ffffe00002320050**. Enter this command.

```
0: kd> !kdexts.devstack ffffe00002320050

  !DevObj           !DrvObj            !DevExt           ObjectName
> ffffe00002320050  \Driver\usbhub     ffffe000023201a0  0000002d
  ffffe0000213c050  \Driver\usbehci    ffffe0000213c1a0  USBPDO-3
...
```

In the preceding output, you can see that the address of the device extension for the FDO of the hub is `ffffe000023201a0`.

Pass the address of the device extension to the **!usbhubext** command.

```
0: kd> !usbkd.usbhubext ffffe000023201a0

FDO ffffe00002320050 PDO ffffe0000213c050 HubNumber# 3
dt USBHUB!_DEVICE_EXTENSION_HUB ffffe000023201a0
!usbhublog ffffe000023201a0
RemoveLock ffffe00002320668
FdoFlags ffffe00002320ba0

CurrentPowerIrp: System (0000000000000000) Device (0000000000000000)
...
## PORT DATA

PortData 1: !port2_info ffffe000021bf000 Port State = PS_WAIT_CONNECT PortChangeLock: 0, Pcq_State:
Pcq_Run_Idle
     PDO 0000000000000000
....
```

In the preceding output, `ffffe000021bf000` is the address of a **_HUB_PORT_DATA** structure. Pass this address to **!usbhubpd**.

```
0: kd> !usbkd.usbhubpd ffffe000021bf000
PortNumber: 1
Parent Hub FDO ffffe00002320050
Device PDO <NULL>
dt USBHUB!_HUB_PORT_DATA ffffe000021bf000
dt USBHUB!_PORTDATA_FLAGS ffffe000021bf968

PortChangelist: !usblist ffffe000021bf1c8, CL [Empty]

## Port Indicators Log (latest at bottom)

##     Event           State               Next

    [EMPTY]

## Port Change Queue History (latest at bottom)

##     Event               State                   Next                    PcqEv_Suspend PcqEv_Resume
PcqEv_ChDone  Tag

01. PCE_Resume           Pcq_Stop                Pcq_Pause                           PcqEv_Reset
PcqEv_Reset   REQUEST_RESUME
...          Pcq_Run_wBusy            Pcq_Run_Idle

## Port Status History (latest at bottom)

##     Current State          Change Eve nt        PDO              CEOSP H/W Port REG Frame Inserted

01. PS_WAIT_CONNECT        REQUEST_PAUSE       0000000000000000 00000 100  Age:000 512498
...
```

See also

USB 2.0 Debugger Extensions

Universal Serial Bus (USB) Drivers

!usbkd.usbhubs

12/5/2018 • 2 minutes to read • Edit Online

The **!usbkd.usbhubs** command displays information about USB hubs.

```
!usbkd.usbhubs a[v]
!usbkd.usbhubs x[v]
!usbkd.usbhubs r[v]
```

Parameters

a
Display all hubs.

r
Display root hubs.

x
Display external hubs.

v
The output is verbose. For example, **!usbhubs rv** displays verbose output about all root hubs.

DLL

Usbkd.dll

Examples

Here is an example of verbose output from the **!usbhubs** command.

```
0: kd> !usbkd.usbhubs rv

args 'rv'
level: r v
ROOT HUBS
*LIST -- Root Hub List @ fffff8000217f440
*flink, blink ffffe000023215c0,ffffe000011f85c0
----------

[0] entry @: ffffe000023201a0
    !DevObj ffffe00002320050 !usbhubext ffffe000023201a0
On Host Controller (0x8086, 0x293c)
        Stat_AsyncResumeStartAt: 2437ee39d29bd498
        Stat_AsyncResumeCompleteAt: 24413c77d29bd498
        Stat_AsyncResume: 0x3c(60) ms
        Stat_SyncResumeStartAt: 2437ee39d29bd498
        Stat_SyncResumeCompleteAt: 2437ee39d29bd498
        Stat_SyncResume: 0x0(0) ms
Trap Regs: Event, Port, Event (ffffe000023204d0)
    Enable: 0 Port: 0 Event 00000000
Hub Number: # 3
Number Of Ports: 4
dt usbhub!_USBHUB_FDO_FLAGS ffffe00002320ba0
>Is Root
>Power Switching:
     No Power Switching
>Overcurrent:
     Global Overcurrent
>PortIndicators:
     No PortIndicators present
>AllowWakeOnConnect:
     DO NOT WakeOnConnect
>CURRENT Hub Wake on Connectstate:
     HWC_DISARM:- do not wake system on connect/disconnect event
>CURRENT Bus Wake state:
     BUS_DISARM:- bus not armed for wake by this hub
>CURRENT Wake Detect state (WW Irp):
     HUB_DISARM:- no ww irp pending (HUB_WAKESTATE_DISARMED)
Milliamps/Port : 500ma
Power caps (0 = not reported)
     PortPower_Registry : 0
     PortPower_DeviceStatus : 500
     PortPower_CfgDescriptor : 500
##      PortPower_HubStatus : 500

----------

[1] entry @: ffffe000008b91a0
    !DevObj ffffe000008b9050 !usbhubext ffffe000008b91a0
On Host Controller (0x8086, 0x2937)
...
```

See also

USB 2.0 Debugger Extensions

Universal Serial Bus (USB) Drivers

!usbkd.usblist

6/25/2019 • 2 minutes to read • Edit Online

The **!usbkd.usblist** command displays a linked list of structures of a specified type.

```
!usbkd.usblist ListAddr, ListType
```

Parameters

ListAddr
Address of a linked list of structures. To find addresses of linked lists maintained by the USB port driver, use **!usbhcdext**. To find addresses of linked list maintained by the USB hub driver, use **!usbhubext**.

ListType
One of the following list types.

LIST TYPE	STRUCTURE
BC	**usbport!_BUS_CONTEXT**
EP	**usbport!_HCD_ENDPOINT**
TT	**usbport!_TRANSACTION_TRANSLATOR**
DL	**usbport!_USBD_DEVICE_HANDLE**
PL	**usbhub!_DEVICE_EXTENSION_PDO**
EL	**usbhub!_HUB_EXCEPTION_RECORD**
RL	**usbhub!_HUB_REFERENCE_LIST_ENTRY**
TL	**usbhub!_HUB_TIMER_OBJECT**
WI	**usbhub!_HUB_WORKITEM**
IO	**usbhub!_IO_LIST_ENTRY**
LA	**usbhub!_LATCH_LIST_ENTRY**
CL	**usbhub!_PORT_CHANGE_CONTEXT**
BL	**usbhub!_SSP_BUSY_HANDLE**

DLL

Usbkd.dll

Examples

Here is one way to find the address of a linked list. First enter **!usbkd.usb2tree**.

```
0: kd> !usbkd.usb2tree
...
2)!ehci_info ffffe00001ca11a0 !devobj ffffe00001ca1050 ...
   ...
```

In the preceding output, the address of the device extension of the FDO is displayed as the argument of the DML command **!ehci_info ffffe00001ca11a0**.

Either click the DML command or pass the address of the device extension to **!usbhcdext**.

```
0: kd> !usbkd.usbhcdext ffffe00001ca11a0

HC Flavor 1000  FDO ffffe00001ca1050
Root Hub: FDO ffffe00002320050 !hub2_info ffffe000023201a0
...
DeviceHandleList: !usblist ffffe00001ca23b8, DL
...
```

In the preceding output, ffffe00001ca23b8 is the address of a linked list of **usbport!_USBD_DEVICE_HANDLE** structures.

Now pass the address of the linked list to **!usblist**.

```
0: kd> !usblist ffffe00001ca23b8, DL
list: ffffe00001ca23b8 DL
----------
!usbdevh ffffe000020f9590
SSP [IdleReady] (0)
PCI\VEN_Xxxx  Xxxx Corporation
Root Hub
DriverName :
----------
!usbdevh ffffe00001bce250
SSP [IdleReady] (0)
USB\Xxxx  Xxxx Corporation
Speed: HIGH, Address:  1, PortPathDepth: 1, PortPath: [3 0 0 0 0 0]
DriverName :\Driver\USBSTOR      !devstack ffffe000053ef2a0
----------
```

See also

USB 2.0 Debugger Extensions

Universal Serial Bus (USB) Drivers

!usbkd.usbpo

12/5/2018 • 2 minutes to read • Edit Online

The **!usbkd.usbpo** command displays the internal list of outstanding USB power requests.

```
!usbkd.usbpo
```

DLL

Usbkd.dll

See also

USB 2.0 Debugger Extensions

Universal Serial Bus (USB) Drivers

!usbkd.usbpdos

12/5/2018 • 2 minutes to read • Edit Online

The **!usbkd.usbpdos** command displays information about all physical device objects (PDOs) created by the USB hub driver.

```
!usbkd.usbpdos
```

DLL

Usbkd.dll

Examples

Here's an example of the output of the **!usbpdos** command.

```
0: kd> !usbkd.usbpdos

ext ffffe00006c513f0
VID 0a12 PID 0001 REV 0309
 dt USBHUB!_USB_DEVICE_DESCRIPTOR ffffe00006c51960
Vendor: Xxxx
 Xxxx Corporation
 dd Class: (224)(e0)
 (224)(e0)
HubFdo: ffffe00000d94050, port: 2
<null>
SystemWake 5 SystemHibernate(S4)
wake_irp_list_head = ffffe00006c51cb0
wake_irp = ffffe00006c51cb0
PDO Times:
    Stat_PdoCreatedAt: 0d29bbd58
    Stat_PdoEnumeratedAt: b65ace75d29bbd58
    Stat: Enumeration Time: 0xa7d3842a(-1479310294) ms
    Stat_Pdo_SetD0_StartAt: 0d29bbd58
    Stat_Pdo_SetD0_CompleteAt: 0d29bbd58
    Stat: PDO Set_D0 time: 0x0(0) ms

##

ext ffffe00007c883f0
VID 0781 PID 5530 REV 0100
 dt USBHUB!_USB_DEVICE_DESCRIPTOR ffffe00007c88960
Vendor: Xxxx
 Xxxx Corporation
 dd Class: (0)(0)
 (8)Class_UsbStorage
HubFdo: ffffe00002320050, port: 3
:Xxxx
SystemWake 5 SystemHibernate(S4)
wake_irp_list_head = ffffe00007c88cb0
wake_irp = ffffe00007c88cb0
PDO Times:
    Stat_PdoCreatedAt: 0d29bbd58
    Stat_PdoEnumeratedAt: 2380af52d29bbd58
    Stat: Enumeration Time: 0x9a24226c(-1708907924) ms
    Stat_Pdo_SetD0_StartAt: 0d29bbd58
    Stat_Pdo_SetD0_CompleteAt: 0d29bbd58
##     Stat: PDO Set_D0 time: 0x0(0) ms
```

See also

USB 2.0 Debugger Extensions

Universal Serial Bus (USB) Drivers

!usbkd.usbpdoxls

12/5/2018 • 2 minutes to read • Edit Online

The **!usbkd.usbpdoxls** command displays information about all physical device objects (PDOs) created by the USB hub driver. For each PDO, this command displays a row of comma-separated values.

```
!usbkd.usbpdoxls
```

DLL

Usbkd.dll

See also

USB 2.0 Debugger Extensions

Universal Serial Bus (USB) Drivers

!usbkd.usbpnp

12/5/2018 • 2 minutes to read • Edit Online

The **!usbkd.usbpnp** command displays state context information about a USB hub.

```
!usbkd.usbpnp DeviceExtension
```

Parameters

DeviceExtension
Address of the device extension for the functional device object (FDO) of a USB hub.

DLL

Usbkd.dll

See also

USB 2.0 Debugger Extensions

Universal Serial Bus (USB) Drivers

!usbkd.usbportisasyncadv

12/5/2018 • 2 minutes to read • Edit Online

The **!usbkd.usbportisasyncadv** command checks all EHCI miniport drivers for an EHCI Interrupt on Async Advance issue.

```
!usbkd.usbportisasyncadv
```

DLL

Usbkd.dll

See also

USB 2.0 Debugger Extensions

Universal Serial Bus (USB) Drivers

!usbkd.usbportmdportlog

12/5/2018 • 2 minutes to read • Edit Online

The **!usbkd.usbportmdportlog** command displays the USBPORT debug log if it is present in a crash dump that was generated as a result of **Bug Check 0xFE**.

```
!usbkd.usbportmdportlog
```

DLL

Usbkd.dll

Remarks

Use this command only when you are debugging a crash dump file that was generated as a result of **Bug Check 0xFE: BUGCODE_USB_DRIVER**.

Examples

Here is an example of a portion of the output of **!usbportmdportlog**.

```
1: kd> !analyze -v
*** ...
BUGCODE_USB_DRIVER (fe)
...
1: kd> !usbkd.usbportmdportlog
Minidump USBPORT DEBUG_LOG buffer size 32768, entries 1024, index 400
*LOG: 0000000113be9600  LOGSTART: 0000000113be6400 *LOGEND: 0000000113bee3e0 # 1024
[ 000] 0000000113be9600 Bfe2 ffffe0001416802c ffffe000020a44f0 ffffe0001416801c
[ 001] 0000000113be9620 Bfe0 0000000000000000 ffffe000039f4720 ffffe00000b76cb0
[ 002] 0000000113be9640 epr+ ffffe000043ee010 ffffe000008f5b80 ffffe00002820a0c
[ 003] 0000000113be9660 alTR ffffe00002820a0c ffffe000039f4720 ffffe00000b76cb0
//
// USBPORT_Core_AllocTransferEx()
// transfer:                ffffe00002820a0c
// Urb:                     ffffe000039f4720
// Irp:                     ffffe00000b76cb0

[ 004] 0000000113be9680 TRcs 0000000000000060 0000000000000468 0000000000000001
[ 005] 0000000113be96a0 urbi ffffe0001422c4cc 0000000000000012 000000000000000b
[ 006] 0000000113be96c0 dURB ffffe000039f4720 ffffe00000b76cb0 000000000000000b
//
// USBPORT_ProcessURB() - Device Handle valid and has been referenced
// Urb:                     ffffe000039f4720
// Irp:                     ffffe00000b76cb0
// function:                URB_FUNCTION_GET_DESCRIPTOR_FROM_DEVICE

[ 007] 0000000113be96e0 vld> ffffe0001421ddd0 0000000000000004 ffffe000039f4720
//
// USBPORT_NeoValidDeviceHandle()
// DeviceHandle:            ffffe0001421ddd0
// ReferenceObj:            ffffe000039f4720

[ 008] 0000000113be9700 devH ffffe0001421ddd0 ffffe000039f4720 0000000000000000
[ 009] 0000000113be9720 pURB ffffe000039f4720 ffffe00000b76cb0 000000000000000b
//
// USBPORT_ProcessURB()
// Urb:                     ffffe000039f4720
// Irp:                     ffffe00000b76cb0
// function:                URB_FUNCTION_GET_DESCRIPTOR_FROM_DEVICE

[ 010] 0000000113be9740 PURB ffffe000039f4720 ffffe000020a44f0 000000000000000b
//
// USBPORT_ProcessURB() - Exit STATUS_PENDING
// Urb:                     ffffe000039f4720
// Irp:                     ffffe000020a44f0
// function:                URB_FUNCTION_GET_DESCRIPTOR_FROM_DEVICE

[ 011] 0000000113be9760 Urb< 0000000000000103 000000000000000b 0000000000000000
[ 012] 0000000113be9780 xSt0 ffffe000012bbf18 0000000000000006 0000000000000001
[ 013] 0000000113be97a0 xnd8 ffffe000012bbf18 ffffe000012bb050 0000000000000000
[ 014] 0000000113be97c0 xnd0 ffffe000012bbf18 ffffe000012bb050 0000000000000000
//
// USBPORT_Xdpc_End()

[ 015] 0000000113be97e0 mapF 0000000000000000 0000000000000000 0000000000000000
[ 016] 0000000113be9800 map5 0000000000000000 0000000000000000 0000000000000000
[ 017] 0000000113be9820 DMAx 0000000000000000 0000000000000000 0000000000000000
[ 018] 0000000113be9840 subx 0000000000000000 0000000000000000 ffffe0001416801c
[ 019] 0000000113be9860 tmoZ 0000000000000000 ffffe0001416801c ffffe000141680a4
...
...
```

See also

USB 2.0 Debugger Extensions

!usbkd.usbportmddcontext

12/5/2018 • 2 minutes to read • Edit Online

The **!usbkd.usbportmddcontext** command displays USBPORT context data if it is present in a crash dump that was generated as a result of **Bug Check 0xFE**.

```
!usbkd.usbportmddcontext
```

DLL

Usbkd.dll

Remarks

Use this command only when you are debugging a crash dump file that was generated as a result of **Bug Check 0xFE: BUGCODE_USB_DRIVER**.

See also

USB 2.0 Debugger Extensions

Universal Serial Bus (USB) Drivers

!usbkd.usbportmddevext

12/5/2018 • 2 minutes to read • Edit Online

The **!usbkd.usbportmddevext** command displays a **usbport!_DEVICE_EXTENSION** structure if one is present in a crash dump that was generated as a result **Bug Check 0xFE**.

```
!usbkd.usbportmddevext
```

DLL

Usbkd.dll

Remarks

Use this command only when you are debugging a crash dump file that was generated as a result of **Bug Check 0xFE: BUGCODE_USB_DRIVER**.

Examples

Here is an example of the output of **!usbportmddevext**.

```
1: kd> !analyze -v
*** ...
BUGCODE_USB_DRIVER (fe)
...
1: kd> !usbkd.usbportmddevext
USBPORT.SYS DEVICE_EXTENSION DATA:
Hcd FDO Extension:
Sig:4f444648 HFDO
CurrentPnpFunc: 0x00000008
PnpFuncHistoryIdx: 0x0000000d
CurrentPowerFunc: 0x00000000
PowerFuncHistoryIdx: 0x00000000
PnpLogIdx: 0x00000002
IoRequestCount: 0x00000007
IoRequestAsyncCallbackCount: 0xffffffff
IoRequestAllow: 0x00000000
Pnp Func History (idx 13)
...
[02] pnp 13 (0d) IRP_MN_FILTER_RESOURCE_REQUIREMENTS
[...
Power Func History (idx 0)
[01] pnp 255 (ff) ??? (x0) PowerDeviceUnspecified
...
    **Power and Wake ----------------------------------------------
    selective suspend:on (1)
    PowerFlags (00000080):
*---FDO---*
PMDebug: 0x00000000
MinAllocedBw: 0x00000000
MaxAllocedBw: 0x00000000
## ...

## XDPC HISTORY_UsbHcIntDpc

State History (idx 2)
EVENT, STATE, NEXT
Log[3] @ 000000d9e7c615cc
Ev_Xdpc_Worker       XDPC_DpcQueued          XDPC_Running
## ...

## XDPC HISTORY_UsbDoneDpc

State History (idx 0)
EVENT, STATE, NEXT
Log[1] @ 000000d9e7c61774
Ev_Xdpc_Worker       XDPC_DpcQueued          XDPC_Running
## ...

## XDPC HISTORY_UsbMapDpc

State History (idx 3)
EVENT, STATE, NEXT
Log[4] @ 000000d9e7c6196c
## ...

## XDPC HISTORY_UsbIocDpc

State History (idx 0)
EVENT, STATE, NEXT
Log[1] @ 000000d9e7c61b04
Ev_Xdpc_Worker       XDPC_DpcQueued          XDPC_Running
...
```

See also

USB 2.0 Debugger Extensions

Universal Serial Bus (USB) Drivers

!usbkd.usbtriage

12/5/2018 • 2 minutes to read • Edit Online

The **!usbkd.usbtriage** command displays a list of USB drivers and device objects.

```
!usbkd.usbtriage
```

DLL

Usbkd.dll

See also

USB 2.0 Debugger Extensions

Universal Serial Bus (USB) Drivers

!usbkd.usbtt

6/25/2019 • 2 minutes to read • Edit Online

The **!usbkd.usbtt** command displays information from a **USBPORT!_TRANSACTION_TRANSLATOR** structure.

```
!usbkd.usbtt StructAddr
```

Parameters

StructAddr
Address of a **usbport!_TRANSACTION_TRANSLATOR** structure. To get the transaction translator list for a USB host controller, use the **!usbkd.usbhcdext** command.

DLL

Usbkd.dll

Examples

Here is one way to find the address of a **usbport!_TRANSACTION_TRANSLATOR** structure. First enter **!usbkd.usb2tree**.

```
0: kd> !usbkd.usb2tree
...
2)!ehci_info ffffe00001ca11a0 !devobj ffffe00001ca1050 PCI: VendorId 8086 DeviceId 293c RevisionId 0002
...
```

In the preceding output, the address of the device extension of the FDO is displayed as the argument of the DML command **!ehci_info ffffe00001ca11a0**.

Either click the DML command or pass the address of the device extension to **!usbhcdext** to get the address of `GlobalTtListHead`. Pass that address to **!usbkd.usblist**, which will display addresses of **_TRANSACTION_TRANSLATOR** structures.

See also

USB 2.0 Debugger Extensions

Universal Serial Bus (USB) Drivers

!usbkd.usbtx

6/25/2019 • 2 minutes to read • Edit Online

The **!usbkd.usbtx** command displays information from a **usbport!_HCD_TRANSFER_CONTEXT** structure.

```
!usbkd.usbtx StructAddr
```

Parameters

StructAddr
Address of a **usbport!_HCD_TRANSFER_CONTEXT** structure. To get the transfer list for a USB host controller, use the **!usbkd.usbhcdext** command.

DLL

Usbkd.dll

Examples

Here is one way to find the address of a **usbport!_HCD_TRANSFER_CONTEXT** structure. First enter **!usbkd.usb2tree**.

```
0: kd> !usbkd.usb2tree
...
4)!uhci_info ffffe00001c8f1a0 !devobj ffffe00001c8f050 PCI: VendorId 8086 DeviceId 2938 RevisionId 0002
...
```

In the preceding output, the address of the device extension of the FDO is displayed as the argument of the DML command **!uhci_info ffffe00001c8f1a0**.

Either click the DML command or pass the address of the device extension to **!usbhcdext** to get the transfer list.

```
0: kd> !usbkd.usbhcdext ffffe00001c8f1a0
...
## I/O TRANSFER LIST(s)

1.) Transfer Request Priority List: (TxQueued) Type: 0-NotSplit, 1-Parent, 2-Child
    ---------------------------------------------------------------------------
    [000]!usbtx ffffe0000653401c !usbep ffffe00004730c60 !irp ffffe00004221220 State: (7)TX_Mapped_inMp
        Priority: 0, Type: 0, Flags= 0000000a, SequenceNum: 10, SplitIdx: 0
        InLen: 4096, OutLen: 0 Status: USBD_STATUS_PENDING (0x40000000)
    ...
```

In the preceding output, `ffffe0000653401c` is the address of an **_HCD_TRANSFER_CONTEXT**structure. Pass this address to **!usbtx**.

```
0: kd> !usbkd.usbtx ffffe0000653401c

dt usbport!_HCD_TRANSFER_CONTEXT ffffe0000653401c
dt usbport!_TRANSFER_PARAMETERS ffffe0000653417c

## TX HISTORY

## EVENT, STATE, NEXT (latest at bottom)

[01]    (23)Ev_TX_Icsq, (0)TX_Undefined, (1)TX_InQueue
[02]    (5)Ev_TX_MapTransfer, (1)TX_InQueue, (2)TX_MapPending
[03]    (7)Ev_TX_MpSubmitSuccess, (2)TX_MapPending, (7)TX_Mapped_inMp

**DMA**
dt usbport!_TRANSFER_SG_LIST ffffe0000653439c
SgCount:  1  MdlVirtualAddress: ffffe00000437000  MdlSystemAddress: ffffe00000437000
    [0] dt usbport!_TRANSFER_SG_ENTRY ffffe000065343bc
    : sysaddr: 0000000000000000 len 0x00001000(4096) offset 0x00000000(0) phys 00000000'ded90000
---
dt usbport!_SCATTER_GATHER_ENTRY ffffe000065343ec
dt _SCATTER_GATHER_LIST ffffe00001bc231c
NumberOfElements = 1
    [0] dt _SCATTER_GATHER_ELEMENT ffffe00001bc232c
     :phys 00000000'ded90000 len 0x00001000(4096)
```

See also

USB 2.0 Debugger Extensions

Universal Serial Bus (USB) Drivers

!usbkd.usbusb2ep

12/5/2018 • 2 minutes to read • Edit Online

The **!usbkd.usbusb2ep** command displays information from a **usbport!_USB2_EP** structure.

```
!usbkd.usbusb2ep StructAddr
```

Parameters

StructAddr
Address of a **usbport!_USB2_EP** structure. To get the address of **usbport!_USB2_EP** structure, use **!usbkd.usb2**.

DLL

Usbkd.dll

See also

USB 2.0 Debugger Extensions

Universal Serial Bus (USB) Drivers

!usbkd.usbusb2tt

12/5/2018 • 2 minutes to read • Edit Online

The **!usbkd.usbusb2tt** command displays information from a **usbport!_TT** structure.

```
!usbkd.usbusb2tt StructAddr
```

Parameters

StructAddr
Address of a **usbport!_TT** structure.

DLL

Usbkd.dll

See also

USB 2.0 Debugger Extensions

Universal Serial Bus (USB) Drivers

!usbkd.usbver

12/5/2018 • 2 minutes to read • Edit Online

The **!usbkd.usbver** command displays the USBD interface version of the USB driver stack.

```
!usbkd.usbver
```

DLL

Usbkd.dll

Remarks

The value of the USBD interface version is stored in the variable `usbport!usbd_version`.

Examples

Here is an example of the output of **!usbkd.usbver**.

```
1: kd> !usbkd.usbver

USBD VER 600 USB stack is VISTA
```

See also

USB 2.0 Debugger Extensions

Universal Serial Bus (USB) Drivers

USBD_IsInterfaceVersionSupported

RCDRKD Extensions

6/25/2019 • 2 minutes to read • Edit Online

This section describes the RCDRKD debugger extension commands. These commands display WPP trace messages created by drivers. Starting with Windows 8, you no longer need a separate trace message format (TMF) file to parse WPP messages. The TMF information is stored in the regular symbol file (PDB file).

Starting in Windows 10, kernel-mode and user-mode drivers can use Inflight Trace Recorder (IFR) for logging traces. Your kernel-mode driver can use the RCDRKD commands to read messages from the circular buffers, format the messages, and display the messages in the debugger.

Note You cannot use the RCDRKD commands to view UMDF driver logs, UMDF framework logs, and KMDF framework logs. To view those logs, use Windows Driver Framework Extensions (Wdfkd.dll) commands.

The RCDRKD debugger extension commands are implemented in Rcdrkd.dll. To load the RCDRKD commands, enter **.load rcdrkd.dll** in the debugger.

The following two commands are the primary commands for displaying trace messages.

- **!rcdrkd.rcdrlogdump**
- **!rcdrkd.rcdrcrashdump**

The following auxiliary commands provide services related to displaying and saving trace messages.

- **!rcdrkd.rcdrloglist**
- **!rcdrkd.rcdrlogsave**
- **!rcdrkd.rcdrsearchpath**
- **!rcdrkd.rcdrsettraceprefix**
- **!rcdrkd.rcdrtmffile**
- **!rcdrkd.rcdrtraceprtdebug**

The **!rcdrkd.rcdrhelp** displays help for the RCDRKD commands in the debugger.

Related topics

WPP Software Tracing

Using the Framework's Event Logger

USB 3.0 Extensions

!rcdrkd.rcdrhelp

12/5/2018 • 2 minutes to read • Edit Online

The **!rcdrkd.rcdrhelp** command displays help for the RCDRKD debugger extension commands.

DLL

Rcdrkd.dll

See also

RCDRKD Extensions

!rcdrkd.rcdrcrashdump

12/5/2018 • 2 minutes to read • Edit Online

The **!rcdrkd.rcdrcrashdump** extension is used with a minidump file to display the recorder log (if the log is present in the minidump).

```
!rcdrkd.rcdrcrashdump TraceProviderGuid
!rcdrkd.rcdrcrashdump DriverName
```

Parameters

TraceProviderGuid
GUID of the trace provider. This parameter must include braces: {*guid*}

DriverName
The name of the driver. The driver name can be used instead of the trace provider GUID for drivers that use the Inflight Trace Recorder (IFR).

DLL

Rcdrkd.dll

See also

RCDRKD Extensions

!rcdrkd.rcdrlogdump

10/4/2019 • 2 minutes to read • Edit Online

The **!rcdrkd.rcdrlogdump** extension displays the trace messages from all recorder buffers of a driver or set of drivers.

```
!rcdrkd.rcdrlogdump DriverName [DriverName ...]
!rcdrkd.rcdrlogdump DriverName -a Address
```

Parameters

DriverName
The name of a driver, not including the .sys extension.

Address
If Address is specified, this command displays the trace messages from the log buffer at the specified address.

DLL

Rcdrkd.dll

Examples

The following example shows a portion of the output of the **!rcdrlogdump**command.

```
3: kd> !rcdrlogdump usbxhci.sys
Trace searchpath is:

Trace format prefix is: %7!u!: %!FUNC! -
Log dump command                          Log ID                   Size
===============                           ======                   ====
!rcdrlogdump  usbxhci -a fffffa8005ff2b60  03 SLT02 DCI04           1024
!rcdrlogdump  usbxhci -a fffffa8005ff2010  03 SLT02 DCI03           1024
!rcdrlogdump  usbxhci -a fffffa8005b36010  03 SLT01 DCI03           1024
!rcdrlogdump  usbxhci -a fffffa8005b379e0  03 SLT01 DCI04           1024
!rcdrlogdump  usbxhci -a fffffa8005b33350  03 SLT02 DCI01           1024
!rcdrlogdump  usbxhci -a fffffa8005b2bb60  03 SLT01 DCI01           1024
!rcdrlogdump  usbxhci -a fffffa8005a2bb60  03 CMD                   1024
!rcdrlogdump  usbxhci -a fffffa8005a1ab60  03 INT00                 1024
!rcdrlogdump  usbxhci -a fffffa8005085330  03 RUNDOWN               512
!rcdrlogdump  usbxhci -a fffffa8005311780  03 1033 0194             1024
Trying to extract TMF information from -
C:\ProgramData\dbg\sym\usbxhci.pdb\D4C85D5D3E2843879EDE226A334D69552\usbxhci.pdb
--- start of log ---
03 RUNDOWN     6: Controller_RetrievePciData - PCI Bus.Device.Function: 48.0.0
03 RUNDOWN     7: Controller_RetrievePciData - PCI: VendorId 0x1033 DeviceId 0x0194 RevisionId 0x03
03 RUNDOWN     8: Controller_QueryDeviceFlagsFromKse - Found DeviceFlags 0x101800 for
USBXHCI:PCI\VEN_1033&DEV_0194
03 RUNDOWN     9: Controller_RetrieveDeviceFlags - DeviceFlags 0x101804
...
03 SLT01 DCI03 89911: TransferRing_DispatchEventsAndReleaseLock - 1.3.0: Mapping Complete : Path 1
TransferRingState_Idle Events 0x00000000
03 SLT01 DCI04 89912: TransferRing_DispatchEventsAndReleaseLock - 1.4.0: Mapping Begin : Path 3
TransferRingState_Mapping Events 0x00000000
03 SLT01 DCI04 89913: TransferRing_DispatchEventsAndReleaseLock - 1.4.0: Mapping Complete : Path 3
TransferRingState_Idle Events 0x00000000
---- end of log ----
```

The preceding output contains messages from several log buffers. To see messages from a single log buffer, use the **-a** parameter, and specify the address of the log buffer. The following example shows how to display the messages from the log buffer at address fffffa8005ff2b60.

```
3: kd> !rcdrlogdump  usbxhci -a fffffa8005ff2b60
Trace searchpath is:

Trace format prefix is: %7!u!: %!FUNC! -
Trying to extract TMF information from -
C:\ProgramData\dbg\sym\usbxhci.pdb\D4C85D5D3E2843879EDE226A334D69552\usbxhci.pdb
--- start of log ---
70914: TransferRing_DispatchEventsAndReleaseLock - 2.4.0: Mapping Complete : Path 1 TransferRingState_Idle
Events 0x00000000
70916: TransferRing_TransferEventHandler - 2.4.0: TransferEventTrb 0xFFFFFA8005A3FBF0 CC_SUCCESS Length 31
EventData 1 Pointer 0xfffffa8005b96210
70917: TransferRing_TransferEventHandler - 2.4.0: WdfRequest 0x0000057FFA469FD8 transferData
0xFFFFFA8005B961B0
70918: TransferRing_TransferComplete - 2.4.0: WdfRequest 0x0000057FFA469FD8 completed with STATUS_SUCCESS
BytesProcessed 31
70922: TransferRing_DispatchEventsAndReleaseLock - 2.4.0: Mapping Begin : Path 3 TransferRingState_Mapping
Events 0x00000000
70923: TransferRing_DispatchEventsAndReleaseLock - 2.4.0: Mapping Complete : Path 3 TransferRingState_Idle
Events 0x00000000
81064: TransferRing_DispatchEventsAndReleaseLock - 2.4.0: Mapping Begin : Path 1 TransferRingState_Mapping
Events 0x00000000
81065: TransferRing_StageRetrieveFromRequest - 2.4.0: WdfRequest 0x0000057FFA469FD8 TransferData
0xFFFFFA8005B961B0 Function 0x9 Length 31 TransferSize 31 BytesProcessed 0
81066: TransferRing_DispatchEventsAndReleaseLock - 2.4.0: Mapping Complete : Path 1 TransferRingState_Idle
Events 0x00000000
81068: TransferRing_TransferEventHandler - 2.4.0: TransferEventTrb 0xFFFFFA8005A40270 CC_SUCCESS Length 31
EventData 1 Pointer 0xfffffa8005b96210
81069: TransferRing_TransferEventHandler - 2.4.0: WdfRequest 0x0000057FFA469FD8 transferData
0xFFFFFA8005B961B0
81070: TransferRing_TransferComplete - 2.4.0: WdfRequest 0x0000057FFA469FD8 completed with STATUS_SUCCESS
BytesProcessed 31
81074: TransferRing_DispatchEventsAndReleaseLock - 2.4.0: Mapping Begin : Path 3 TransferRingState_Mapping
Events 0x00000000
81075: TransferRing_DispatchEventsAndReleaseLock - 2.4.0: Mapping Complete : Path 3 TransferRingState_Idle
Events 0x00000000
---- end of log ----
```

See also

RCDRKD Extensions

!rcdrkd.rcdrloglist

12/5/2018 • 2 minutes to read • Edit Online

The **!rcdrkd.rcdrloglist** extension displays a list of the recorder logs owned by a driver or a set of drivers.

```
!rcdrkd.rcdrloglist DriverName [DriverName ...]
```

Parameters

DriverName
The name of a driver, not including the .sys extension.

DLL

Rcdrkd.dll

Remarks

This command is relevant only for drivers that log messages to different logs by using the WppRecorder API.

Examples

The following example displays a list of all recorder logs owned by the USB 3.0 host controller driver (usbxhci.sys).

```
3: kd> !rcdrloglist usbxhci
Log dump command                          Log ID                  Size
================                          ======                  ====
!rcdrlogdump  usbxhci -a fffffa8005ff2b60  03 SLT02 DCI04          1024
!rcdrlogdump  usbxhci -a fffffa8005ff2010  03 SLT02 DCI03          1024
!rcdrlogdump  usbxhci -a fffffa8005b36010  03 SLT01 DCI03          1024
!rcdrlogdump  usbxhci -a fffffa8005b379e0  03 SLT01 DCI04          1024
!rcdrlogdump  usbxhci -a fffffa8005b33350  03 SLT02 DCI01          1024
!rcdrlogdump  usbxhci -a fffffa8005b2bb60  03 SLT01 DCI01          1024
!rcdrlogdump  usbxhci -a fffffa8005a2bb60  03 CMD                  1024
!rcdrlogdump  usbxhci -a fffffa8005a1ab60  03 INT00                1024
!rcdrlogdump  usbxhci -a fffffa8005085330  03 RUNDOWN              512
!rcdrlogdump  usbxhci -a fffffa8005311780  03 1033 0194            1024
```

See also

RCDRKD Extensions

!rcdrkd.rcdrlogsave

12/5/2018 • 2 minutes to read • Edit Online

The **!rcdrkd.rcdrlogsave** extension saves the recorder buffers of a driver.

```
!rcdrkd.rcdrlogsave DriverName [CaptureFilename ]
```

Parameters

DriverName
The name of the driver, not including the .sys extension.

CaptureFileName
The name of the file (not including the .etl extension) in which to save the recorder buffers. If *CaptureFileName* is not specified, the recorder buffers are saved in *DriverName*.etl.

DLL

Rcdrkd.dll

See also

RCDRKD Extensions

!rcdrkd.rcdrsearchpath

12/5/2018 • 2 minutes to read • Edit Online

The **!rcdrkd.rcdrsearchpath** extension sets the search path for trace message format (TMF) and trace message control (TMC) files.

```
!rcdrkd.rcdrsearchpath FilePath
```

Parameters

FilePath
Path to the format files.

DLL

Rcdrkd.dll

Remarks

The search path set by this command takes precedence over the search path specified in the TRACE_FORMAT_SEARCH_PATH environment variable.

See also

RCDRKD Extensions

!rcdrkd.rcdrsettraceprefix

6/25/2019 • 2 minutes to read • Edit Online

The **!rcdrkd.rcdrsettraceprefix** extension sets the trace message prefix.

```
!rcdrkd.rcdrsettraceprefix TracePrefixString
```

Parameters

TracePrefixString
The trace message prefix string.

DLL

Rcdrkd.dll

Remarks

Each message in a recorder log has a prefix that you can control by specifying a trace message prefix string. For more information, see Trace Message Prefix.

Examples

In the following example, the trace message prefix is originally **%7!u!: %!FUNC! -** . The parameter **%7!u!** specifies that the prefix includes the message sequence number. The parameter **%!FUNC!** specifies that the prefix includes the name of the function that generated the message. The example calls **!rcdrsettraceprefix** to change the prefix string to **%7!u!**. After that, the log display includes message sequence numbers, but does not include function names.

```
0: kd> !rcdrlogdump USBXHCI -a 0xfffffa8010737b60
Trace searchpath is:

Trace format prefix is: %7!u!: %!FUNC! -
Trying to extract TMF information from -
C:\ProgramData\dbg\sym\usbxhci.pdb\D4C85D5D3E2843879EDE226A334D69552\usbxhci.pdb
--- start of log ---
405: TransferRing_DispatchEventsAndReleaseLock - 1.8.1: Mapping Begin : Path 1 TransferRingState_Mapping
Events 0x00000000
406: TransferRing_StageRetrieveFromRequest - 1.8.1: WdfRequest 0x0000057FEE117A88 TransferData
0xFFFFFA8011EE8700 Function 0x9 Length 500 TransferSize 500 BytesProcessed 0
...
---- end of log ----

0: kd> !rcdrsettraceprefix %7!u!:
SetTracePrefix: "%7!u!: "
0: kd> !rcdrlogdump USBXHCI -a 0xfffffa8010737b60
Trace searchpath is:

Trace format prefix is: %7!u!:
Trying to extract TMF information from -
C:\ProgramData\dbg\sym\usbxhci.pdb\D4C85D5D3E2843879EDE226A334D69552\usbxhci.pdb
--- start of log ---
405: 1.8.1: Mapping Begin : Path 1 TransferRingState_Mapping Events 0x00000000
406: 1.8.1: WdfRequest 0x0000057FEE117A88 TransferData 0xFFFFFA8011EE8700 Function 0x9 Length 500 TransferSize
500 BytesProcessed 0
...
---- end of log ----
```

See also

RCDRKD Extensions

!rcdrkd.rcdrtmffile

12/5/2018 • 2 minutes to read • Edit Online

The **!rcdrkd.rcdrtmffile** extension sets or clears the name of the trace message format (TMF) file.

```
!rcdrkd.rcdrtmffile [Filename]
```

Parameters

Filename
The name of the TMF file. If this parameter is not specified, the filename is cleared.

DLL

Rcdrkd.dll

See also

RCDRKD Extensions

!rcdrkd.rcdrtraceprtdebug

12/5/2018 • 2 minutes to read • Edit Online

The **!rcdrkd.rcdrtraceprtdebug** extension turns TracePrt diagnostic mode on or off. This extension should be used under the direction of support.

```
!rcdrkd.rcdrtraceprtdebug {on|off}
```

Parameters

on
Turns TracePrt diagnostic mode on.

off
Turns TracePrt diagnostic mode off.

DLL

Rcdrkd.dll

See also

RCDRKD Extensions

HID Extensions

3/5/2019 • 2 minutes to read • Edit Online

This section describes the Human Interface Device (HID) debugger extension commands.

The HID debugger extension commands are implemented in Hidkd.dll. To load the HID commands, enter **.load hidkd.dll** in the debugger.

Getting started with the HID extensions

To start debugging a HID issue, enter the **!hidtree** command. The **!hidtree** command displays a list of commands and addresses that you can use to investigate device objects, preparsed HID data, and HID report descriptors.

In this section

TOPIC	DESCRIPTION
!hidkd.help	The **!hidkd.help** command displays help for the HID debugger extension commands.
!hidkd.hidfdo	The **!hidkd.hidfdo** command displays HID information associated with a functional device object (FDO).
!hidkd.hidpdo	The **!hidkd.hidpdo** command displays HID information associated with a physical device object (PDO).
!hidkd.hidtree	The **!hidkd.hidtree** command displays a list of all device nodes that have a HID function driver along with their child nodes. The child nodes have a physical device object (PDO) that was created by the parent node's HID function driver.
!hidkd.hidppd	The **!hidkd.hidppd** command displays HID preparsed data.
!hidkd.hidrd	The **!hidkd.hidrd** command displays a HID report descriptor in both raw and parsed format.

Related topics

RCDRKD Extensions

Specialized Extension Commands

!hidkd.help

12/5/2018 • 2 minutes to read • Edit Online

The **!hidkd.help** command displays help for the HID debugger extension commands.

DLL

Hidkd.dll

See also

HID Extensions

!hidkd.hidfdo

12/5/2018 • 2 minutes to read • Edit Online

The **!hidkd.hidfdo** command displays HID information associated with a functional device object (FDO).

```
!hidkd.hidfdo fdo
```

Parameters

fdo
Address of an FDO. To get the addresses of FDOs that are associated with HID drivers, use the **!usbhid.hidtree** command.

DLL

Hidkd.dll

Examples

Here is an example of the output of the **!hidfdo** command. The example first calls **!hidtree** to get the address of an FDO.

```
0: kd> !hidkd.hidtree
HID Device Tree
...
FDO  VendorID:0x045E(Microsoft Corporation) ProductID:0x0745 Version:0x0634
!hidfdo 0xffffe00004f466e0
...
0: kd> !hidfdo 0xffffe00004f466e0
# FDO 0xffffe00004f466e0  (!devobj/!devstack)

  Name              : \Device\_HID00000002
  Vendor ID         : 0x045E(Microsoft Corporation)
  Product ID        : 0x0745
  Version Number    : 0x0634
  Is Present?       : Y
  Report Descriptor : !hidrd 0xffffe00004281a80 0x127
  Per-FDO IFR Log   : !rcdrlogdump HIDCLASS -a 0xFFFFE0000594D000

  Position in HID tree

  dt FDO_EXTENSION 0xffffe00004f46850
```

See also

HID Extensions

!hidkd.hidpdo

12/5/2018 • 2 minutes to read • Edit Online

The **!hidkd.hidpdo** command displays HID information associated with a physical device object (PDO).

```
!hidkd.hidpdo pdo
```

Parameters

pdo
Address of a PDO. To get the addresses of PDOs that are associated with HID drivers, use the **!usbhid.hidtree** command.

DLL

Hidkd.dll

Examples

Here is an example of the output of the **!hidpdo** command. The example first calls **!hidtree** to get the address of a PDO.

```
0: kd> !hidkd.hidtree
HID Device Tree
...
FDO  VendorID:0x045E(Microsoft Corporation) ProductID:0x0745 Version:0x0634
...
    PDO  Generic Desktop Controls (0x01) | Mouse (0x02)
    !hidpdo 0xffffe000056281e0
    ...

0: kd> !hidpdo 0xffffe000056281e0
# PDO 0xffffe000056281e0  (!devobj/!devstack)

  Collection Num  : 1
  Name            : \Device\_HID00000002#COLLECTION00000001
  FDO             : !hidfdo 0xffffe00004f466e0
  Per-FDO IFR Log : !rcdrlogdump HIDCLASS -a 0xFFFFE0000594D000

  Usage Page      : Generic Desktop Controls (0x01)
  Usage           : Mouse (0x02)
  Report Length   : 0xa(Input) 0x0(Output) 0x2(Feature)
  Pre-parsed Data : 0xffffe00003742840
  Position in HID tree

  dt PDO_EXTENSION 0xffffe00005628350

## Power States

  Power States  : S0/D0
  Wait Wake IRP : !irp 0xffffe00004fc57d0 (pending on \Driver\HidUsb)
```

See also

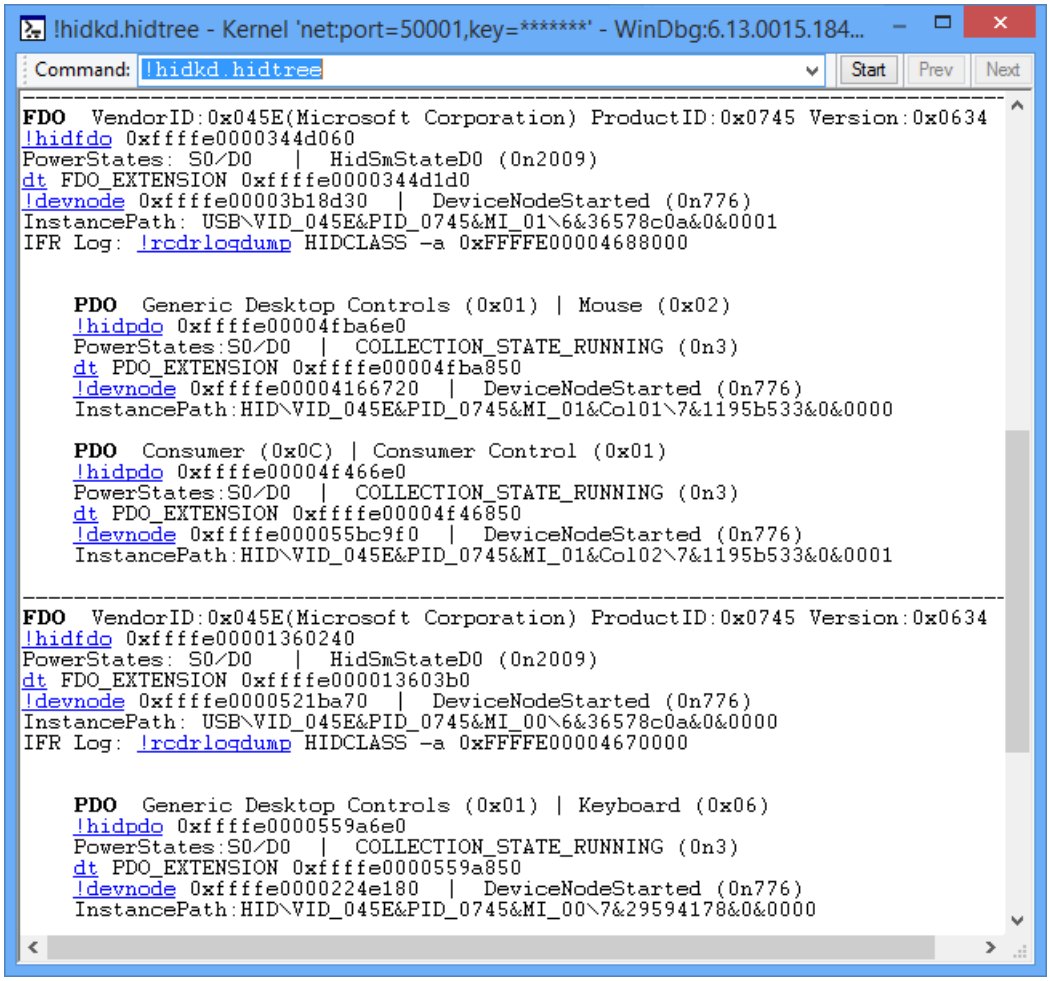
!hidkd.hidtree

12/5/2018 • 2 minutes to read • Edit Online

The **!hidkd.hidtree** command displays a list of all device nodes that have a HID function driver along with their child nodes. The child nodes have a physical device object (PDO) that was created by the parent node's HID function driver.

```
!hidkd.hidtree
```

This screen shot shows an example of the output of the **!hidtree** command.



O

In this example, there are two device nodes that have a HID function driver. A functional device object (FDO) represents the HID driver in those two nodes. The first FDO node has two child nodes, and the second FDO node has one child node. In the debugger output, the child nodes have the PDO heading.

Note This set of device nodes does not form a tree that has a single root node. The device nodes that have HID function drivers can be isolated from each other.

When you are debugging a HID issue, the **!hidtree** is a good place to start, because the command displays several addresses that you can pass to other HID debugger commands. The output uses Debugger Markup Language (DML) to provide links. The links execute commands that give detailed information related to an individual device node. For example, you could get information about an FDO by clicking one of the **!hidfdo** links. As an alternative to clicking a link, you can enter a command. For example, to see detailed information

about the first node in the preceding output, you could enter the command **!devnode 0xffffe00003b18d30**.

Note The DML feature is available in WinDbg, but not in Visual Studio or KD.

DLL

Hidkd.dll

See also

HID Extensions

!hidkd.hidppd

12/5/2018 • 2 minutes to read • Edit Online

The **!hidkd.hidppd** command displays HID preparsed data.

```
!hidkd.hidppd ppd
```

Parameters

ppd
Address of a **HIDP_PREPARSED_DATA** structure. To get the address of a **HIDP_PREPARSED_DATA** structure, use **!hidfdo** or **!hidpdo**.

DLL

Hidkd.dll

Examples

This example shows how to use **!hidpdo** followed by **!hidppd**. The output of **!hidpdo** shows the address of a **HIDP_PREPARSED_DATA** structure.

```
0: kd> !hidpdo 0xffffe000029f6060
## PDO 0xffffe000029f6060  (!devobj/!devstack)

  Collection Num  : 1
  Name            : \Device\_HID00000000#COLLECTION00000001
  ...
  Pre-parsed Data : 0xffffe000029d1010
  ...

0: kd> !hidkd.hidppd 0xffffe000029d1010
Reading preparsed data...
Preparsed Data at 0xffffe000029d1010

## Summary

  UsagePage             : Vendor-defined (0xFFA0)
  Usage                 : 0x01
  Report Length         : 0x2(Input) 0x2(Output) 0x0(Feature)
  Link Collection Nodes : 2
  Button Caps           : 0(Input) 0(Output) 0(Feature)
  Value Caps            : 1(Input) 1(Output) 0(Feature)
  Data Indices          : 1(Input) 1(Output) 0(Feature)

## Input Value Capability #0

  Report ID         : 0x0
  Usage Page        : Vendor-defined (0xFFA1)
  Bit Size          : 0x8
  Report Count      : 0x1
  ...
```

See also

HID Extensions

!hidkd.hidrd

12/5/2018 • 2 minutes to read • Edit Online

The **!hidkd.hidrd** command displays a HID report descriptor in both raw and parsed format.

```
!hidkd.hidrd rd Length
```

Parameters

rd
Address of the raw report descriptor data. To get the address of the descriptor data, use the **!hidfdo** command.

Length
The length, in bytes, of the raw report descriptor data. To get the length, use the **!hidfdo** command.

DLL

Hidkd.dll

Examples

This example shows how to use the **!hidfdo** command followed by the **!hidrd** command. The output of **!hidfdo** shows both the address and length of the raw report descriptor data.

```
0: kd> !hidfdo 0xffffe00004f466e0
# FDO 0xffffe00004f466e0  (!devobj/!devstack)

  Name              : \Device\_HID00000002
  ...
  Report Descriptor : !hidrd 0xffffe00004281a80 0x127
  ...

0: kd> !hidrd 0xffffe00004281a80 0x127
Report Descriptor at 0xffffe00004281a80

## Raw Data

0x0000: 05 01 09 02 A1 01 05 01-09 02 A1 02 85 1A 09 01
0x0010: A1 00 05 09 19 01 29 05-95 05 75 01 15 00 25 01
0x0020: 81 02 75 03 95 01 81 01-05 01 09 30 09 31 95 02
...

## Parsed

Usage Page (Generic Desktop Controls)...................0x0000: 05 01
Usage (Mouse)...........................................0x0002: 09 02
Collection (Application).................................0x0004: A1 01
..Usage Page (Generic Desktop Controls).................0x0006: 05 01
..Usage (Mouse).........................................0x0008: 09 02
..Collection (Logical)..................................0x000A: A1 02
....Report ID (26)......................................0x000C: 85 1A
...
End Collection ()......................................0x0126: C0
```

See also

HID Extensions

Logger Extensions (Logexts.dll)

12/5/2018 • 2 minutes to read • Edit Online

Extension commands related to the Logger and LogViewer tools can be found in Logexts.dll.

This DLL appears in the winext directory. It can be used with all Windows operating systems.

!logexts.help

12/5/2018 • 2 minutes to read • Edit Online

The **!logexts.help** extension displays a Help text in the Command Prompt window showing all Logexts.dll extension commands.

```
!logexts.help
```

DLL

Windows 2000	Logexts.dll
Windows XP and later	Logexts.dll

Additional Information

For more information, see Logger and LogViewer.

!logexts.logb

12/5/2018 • 2 minutes to read • Edit Online

The **!logexts.logb** extension displays or flushes the output buffer.

```
!logexts.logb p
!logexts.logb f
```

Parameters

p
Causes the contents of the output buffer to be displayed in the debugger.

f
Flushes the output buffer to the disk.

DLL

Windows 2000	Logexts.dll
Windows XP and later	Logexts.dll

Additional Information

For more information, see Logger and LogViewer.

Remarks

As a performance consideration, log output is flushed to disk only when the output buffer is full. By default, the buffer is 2144 bytes.

The **!logexts.logb p** extension displays the contents of the buffer in the debugger.

The **!logexts.logb f** extension flushes the buffer to the log files. Because the buffer memory is managed by the target application, the automatic writing of the buffer to disk will not occur if there is an access violation or some other nonrecoverable error in the target application. In such cases, you should use this command to manually flush the buffer to the disk. Otherwise, the most recently-logged APIs might not appear in the log files.

!logexts.logc

12/5/2018 • 2 minutes to read • Edit Online

The **!logexts.logc** extension displays all API categories, displays all APIs in a specific category, or enables and disables the logging of APIs in one or more categories.

```
!logexts.logc e Categories
!logexts.logc d Categories
!logexts.logc p Category
!logexts.logc
```

Parameters

e
Enables logging of the specified categories.

d
Disables logging of the specified categories.

Categories
Specifies the categories to be enabled or disabled. If multiple categories are listed, separate them with spaces. An asterisk (*) can be used to indicate all categories.

p
Displays all APIs that belong to the specified category.

Category
Specifies the category whose APIs will be displayed. Only one category can be specified with the **p** option.

DLL

Windows 2000	Logexts.dll
Windows XP and later	Logexts.dll

Additional Information

For more information, see Logger and LogViewer.

Remarks

Without any options, **!logexts.logc** will display the current list of available categories and will indicate which ones are enabled and disabled.

If a category is disabled, the hooks for all APIs in that category will be removed so there is no longer any performance overhead. COM hooks are not removed, because they cannot be re-enabled at will.

Enabling only certain categories can be useful when you are only interested in a particular type of interaction that the program is having with Windows (for example, file operations). This reduces the log file size and also reduces the effect that Logger has on the execution speed of the process.

The following command will enable the logging of all categories:

```
0:000> !logexts.logc e *
```

The following command will disable the logging of category 7:

```
0:000> !logexts.logc d 7
```

The following command will enable the logging of categories 13 and 15:

```
0:000> !logexts.logc e 13 15
```

The following command will display all APIs belonging to category 3:

```
0:000> !logexts.logc p 3
```

!logexts.logd

12/5/2018 • 2 minutes to read • Edit Online

The **!logexts.logd** extension disables logging.

```
!logexts.logd
```

DLL

Windows 2000	Logexts.dll
Windows XP and later	Logexts.dll

Additional Information

For more information, see Logger and LogViewer.

Remarks

This will cause all API hooks to be removed in an effort to allow the program to run freely. COM hooks are not removed, because they cannot be re-enabled at will.

!logexts.loge

12/5/2018 • 2 minutes to read • Edit Online

The **!logexts.loge** extension enables logging. If logging has not been initialized, it will be initialized and enabled.

```
!logexts.loge [OutputDirectory]
```

Parameters

OutputDirectory
Specifies the directory to use for output. If *OutputDirectory* is specified, it must exist -- the debugger will not create it. If a relative path is specified, it will be relative to the directory from which the debugger was started. If *OutputDirectory* is omitted, the path to the Desktop will be used. The debugger will create a LogExts subdirectory of *OutputDirectory*, and all Logger output will be placed in this subdirectory.

DLL

Windows 2000	Logexts.dll
Windows XP and later	Logexts.dll

Additional Information

For more information, see Logger and LogViewer.

Remarks

If Logger has not yet been injected into the target application by the **!logexts.logi** extension, the **!logexts.loge** extension will inject Logger into the target and then enable logging.

If **!logexts.logi** has already been run, you cannot include *OutputDirectory*, because the output directory will have already been set.

!logexts.logi

12/5/2018 • 2 minutes to read • Edit Online

The **!logexts.logi** extension initializes logging by injecting Logger into the target application.

```
!logexts.logi [OutputDirectory]
```

Parameters

OutputDirectory
Specifies the directory to use for output. If *OutputDirectory* is specified, it must exist -- the debugger will not create it. If a relative path is specified, it will be relative to the directory from which the debugger was started. If *OutputDirectory* is omitted, the path to the Desktop will be used. The debugger will create a LogExts subdirectory of *OutputDirectory*, and all Logger output will be placed in this subdirectory.

DLL

Windows 2000	Logexts.dll
Windows XP and later	Logexts.dll

Additional Information

For more information, see Logger and LogViewer.

Remarks

This command initializes logging, but does not actually enable it. Logging can be enabled with the **!logexts.loge** command.

!logexts.logm

12/5/2018 • 2 minutes to read • Edit Online

The **!logexts.logm** extension creates or displays a module inclusion list or a module exclusion list.

```
!logexts.logm i Modules
!logexts.logm x Modules
!logexts.logm
```

Parameters

i

Causes Logger to use a module inclusion list. It will consist of the specified *Modules*.

x

Causes Logger to use a module exclusion list. It will consist of Logexts.dll, kernel32.dll, and the specified *Modules*.

Modules

Specifies the modules to be included or excluded. This list is not cumulative; each use of this command creates an entirely new list. If multiple modules are listed, separate them with spaces. An asterisk (*) can be used to indicate all modules.

DLL

Windows 2000	Logexts.dll
Windows XP and later	Logexts.dll

Additional Information

For more information, see Logger and LogViewer.

Remarks

With no parameters, the **!logexts.logm** extension displays the current inclusion list or exclusion list.

The extensions **!logexts.logm x *** and **!logexts.logm i** are equivalent: they result in a completely empty inclusion list.

The extensions **!logexts.logm i *** and **!logexts.logm x** are equivalent: they result in an exclusion list that contains only Logexts.dll and kernel32.dll. These two modules are always excluded, because Logger is not permitted to log itself.

Here are some examples:

```
0:001> !logm
Excluded modules:
  LOGEXTS.DLL      [mandatory]
  KERNEL32.DLL     [mandatory]
  USER32.DLL
  GDI32.DLL
  ADVAPI32.DLL

0:001> !logm x winmine.exe
Excluded modules:
  Logexts.dll      [mandatory]
  kernel32.dll     [mandatory]
  winmine.exe

0:001> !logm x user32.dll gdi32.dll
Excluded modules:
  Logexts.dll      [mandatory]
  kernel32.dll     [mandatory]
  user32.dll
  gdi32.dll

0:001> !logm i winmine.exe mymodule2.dll
Included modules:
  winmine.exe
  mymodule2.dll
```

!logexts.logo

12/5/2018 • 2 minutes to read • Edit Online

The **!logexts.logo** extension sets or displays the Logger output options.

```
!logexts.logo {e|d} {d|t|v}
!logexts.logo
```

Parameters

e|d
Specifies whether to enable (e) or disable (d) the indicated output type.

d|t|v
Specifies the output type. Three types of Logger output are possible: messages sent directly to the debugger (d), a text file (t), or a verbose .lgv file (v).

DLL

Windows 2000	Logexts.dll
Windows XP and later	Logexts.dll

Additional Information

For more information, see Logger and LogViewer.

Remarks

If **!logexts.logo** is used without any parameters, then the current logging status, the output directory, and the current settings for the debugger, text file, and verbose log are displayed:

```
0:000> !logo
Logging currently enabled.

Output directory: MyLogs\LogExts\

Output settings:
  Debugger            Disabled
  Text file           Enabled
  Verbose log         Enabled
```

In the previous example, the output directory is a relative path, so it is located relative to the directory in which the debuggers were started.

To disable verbose logging, you would use the following command:

```
0:000> !logo d v
  Debugger            Disabled
  Text file           Enabled
  Verbose log         Disabled
```

Text file and .lgv files will be placed in the current output directory. To read an .lgv file, use LogViewer.

NDIS Extensions (Ndiskd.dll)

10/23/2019 • 3 minutes to read • Edit Online

This section describes commands available in !ndiskd, a debugger extension that is useful for debugging NDIS (Network Device Interface Specification) drivers. These commands enable network driver developers to see a bigger picture of the Windows networking stack and how their drivers interact with it. With !ndiskd, you can see the state of all network adapters (**!ndiskd.netadapter**), a visual diagram of the computer's network stack (**!ndiskd.netreport**), a log of traffic on the network adapters(**!ndiskd.nbllog**), or a list of all pending OID requests (**!ndiskd.oid**).

The commands can be found in Ndiskd.dll. To load the symbols, enter **.reload /f ndis.sys** in the debugger command window. To confirm the symbols loaded successfully, use the **!lmi ndis** extension and look for the phrase "Symbols loaded successfully" toward the bottom. Your output should look similar to the following example:

```
3: kd> !lmi ndis
Loaded Module Info: [ndis]
         Module: ndis
   Base Address: fffff80e2e150000
     Image Name: ndis.sys
   Machine Type: 34404 (X64)
     Time Stamp: 57f4c58d Wed Oct  5 02:19:09 2016
           Size: 128000
       CheckSum: 1213f7
Characteristics: 22
Debug Data Dirs: Type  Size     VA  Pointer
             CODEVIEW    21, 7b1b8,   79fb8 RSDS - GUID: {06A2E58C-996D-4419-81A0-BB293BD63BCA}
               Age: 1, Pdb: ndis.pdb
                 ??   880, 7b1dc,   79fdc [Data not mapped]
     Image Type: MEMORY   - Image read successfully from loaded memory.
    Symbol Type: PDB      - Symbols loaded successfully from image header.
                 c:\Debuggers\sym\ndis.pdb\06A2E58C996D441981A0BB293BD63BCA1\ndis.pdb
       Compiler: Linker - front end [0.0 bld 0] - back end [14.0 bld 23917]
    Load Report: private symbols & lines, source indexed
                 c:\Debuggers\sym\ndis.pdb\06A2E58C996D441981A0BB293BD63BCA1\ndis.pdb
```

!ndiskd Hyperlinks

Many of the extension commands in !ndiskd present you with hyperlinks in the results they display in the debugger window. The text for these hyperlinks has been left in the samples provided to illustrate the exact format of what you will see when you run the command on your debugee machine. Some of the examples also refer explicitly to clicking on these links so you can understand typical usage flows, though the examples also provide the alternate command line forms of each command.

Common Parameters

All !ndiskd commands support the following generic parameters.

-verbose
Shows additional details.

-terse
Suppresses some boilerplate output.

-static
Suppresses some interactive output.

-dml 0|1
Controls whether DML (debugger markup language) output is enabled.

-unicode 0|1
Controls whether Unicode character output is allowed.

-indent N
Uses *N* spaces per level of indent.

-force
Overrides some safety checks on remote data sanity.

-tracedata
Shows verbose trace messages to debug !ndiskd itself.

Net Adapter, NDIS Driver, and General Commands

The following commands display information about the machine's network adapters, network drivers, and general commands associated with the network stack (such as rcvqueues, opens, filters, OIDs, and RW locks).

- **!ndiskd.netadapter**
- **!ndiskd.minidriver**
- **!ndiskd.rcvqueue**
- **!ndiskd.protocol**
- **!ndiskd.mopen**
- **!ndiskd.filter**
- **!ndiskd.filterdriver**
- **!ndiskd.oid**
- **!ndiskd.ndisrwlock**
- **!ndiskd.netreport**

NET_BUFFER_LIST and NET_BUFFER Commands

The following commands display information relating to **NET_BUFFER_LIST** and **NET_BUFFER** structures.

- **!ndiskd.nbl**
- **!ndiskd.nb**
- **!ndiskd.nblpool**
- **!ndiskd.nbpool**
- **!ndiskd.pendingnbls**
- **!ndiskd.nbllog**

NetAdapterCx Commands

The following commands display information relating to the Network Adapter WDF Class Extension (NetAdapterCx)[LINK TBD] and its associated structures, NET_RING_BUFFER [LINK TBD] and NET_PACKET [LINK TBD].

- **!ndiskd.cxadapter**
- **!ndiskd.netqueue**
- **!ndiskd.netrb**

- **!ndiskd.netpacket**
- **!ndiskd.netpacketfragment**

Network Interface Commands

The following commands display information relating to network interfaces.

- **!ndiskd.interfaces**
- **!ndiskd.ifprovider**

NDIS_PACKET Commands

The following commands display information about NDIS_PACKET structures. These extensions are for legacy NDIS 5.x drivers. The NDIS_PACKET structure and its associated architecture have been deprecated.

- **!ndiskd.pkt**
- **!ndiskd.pktpools**
- **!ndiskd.findpacket**

CoNDIS Commands

The following commands display information about Connection-Oriented NDIS connections.

- **!ndiskd.vc**
- **!ndiskd.af**

NDIS Debugging Commands

The following commands display information relating to NDIS refcounts, event logs, stack traces, and debug traces.

- **!ndiskd.ndisref**
- **!ndiskd.ndisevent**
- **!ndiskd.ndisstack**
- **!ndiskd.dbglevel**
- **!ndiskd.dbgsystems**

WDI Commands

The following commands display information about WDI Miniport Drivers.

- **!ndiskd.wdiadapter**
- **!ndiskd.wdiminidriver**
- **!ndiskd.nwadapter**

NDIS and !ndiskd Information Commands

The following commands display information about NDIS.sys and ndiskd.dll.

- **!ndiskd.ndis**
- **!ndiskd.ndiskdversion**

Miscellaneous Commands

- **!ndiskd.ifstacktable**
- **!ndiskd.compartments**
- **!ndiskd.ndisslot**

Related Topics

For more information about designing NDIS drivers for Windows Vista and later, see the Network Driver Design Guide.

For more information about reference for NDIS drivers for Windows Vista and later, see Windows Vista and Later Networking Reference.

For a demonstration of using the !ndiskd debugger commands to debug the network stack, see Debugging the Network Stack.

!ndiskd.help

10/23/2019 • 2 minutes to read • Edit Online

The **!ndiskd.help** command displays a list of available !ndiskd commands with a brief description of each one.

```
!ndiskd.help
```

DLL

Ndiskd.dll

Examples

The following example shows the list of help commands using **!ndiskd.help**.

```
3: kd> !ndiskd.help

NDIS KD EXTENSIONS

    help               This help and lots more
    netadapter         Show network adapters  (this is a good starting place)
    protocol           Show protocol drivers
    mopen              Show open bindings between netadapter and protocol
    filter             Show light-weight filters
    nbl                Show information about an NET_BUFFER_LIST
    oid                Show pending OID Requests
    ndis               Show NDIS.sys build info
    netreport          Draw a box diagram of your network stack

    Show more extensions
    View examples & tutorials online
```

By using **!ndiskd.help -all**, you'll get a more detailed list, as shown in the following example.

Note
Some alternate commands are listed at the bottom of this example. These commands are available for NDIS driver developers who have used them before but we recommend using the primary commands instead.

```
3: kd> !ndiskd.help -all

NDIS KD EXTENSIONS

    Primary commands:
    help               This help and lots more
    netadapter         Show network adapters  (this is a good starting place)
        minidriver     Show network adapter drivers
        rcvqueue       Show a receive queue (c.f. VMQ)
    protocol           Show protocol drivers
    mopen              Show open bindings between netadapter and protocol
    filter             Show light-weight filters
        filterdriver   Show filter drivers (not to be confused with filters)
    nbl                Show information about an NET_BUFFER_LIST
    nb                 Show information about an NET_BUFFER
    nblpool            Show information about an NET_BUFFER_LIST pool
    nbpool             Show information about an NET_BUFFER pool
    pendingnbls        Show all NET_BUFFER_LISTs that are in transit
    nbllog             Show a log of all NET_BUFFER_LIST activity
    oid                Show pending OID Requests
    interfaces         Show interfaces (à la NDISIF)
        ifprovider     Show registered NDIS interface providers
        ifstacktable   Show the ifStackTable
        networks       Show networks (probably not what you think)
        compartments   Show compartments
    pkt                Show a NDIS_PACKET structure
        pktpools       Show all allocated packet pools
        findpacket     Find a packet in memory
    vc                 Show a CoNDIS virtual connection
    af                 Show a CoNDIS address family
    ndisref            Show a debug log of refcounts
    ndisevent          Show a debug log of events
    ndisstack          Show a debug stack trace
    wdiadapter         Shows one or more WDIWiFi!CAdapter structures
    wdiminidriver      Shows one or more CMiniportDriver structures
    nwadapter          Shows one or more nwifi!ADAPT structures
    ndisrwlock         Show an NDIS_RW_LOCK_EX lock
    ndisslot           Show an NDIS per-processor slot
    ndis               Show NDIS.sys build info
        dbglevel       Change the debugging level [checked NDIS.sys only]
        dbgsystems     Toggle subsystems being debugged [checked NDIS.sys only]
        ndiskdversion  Show info about NDISKD itself
    netreport          Draw a box diagram of your network stack

    Alternate commands:
    miniport           Same as !ndiskd.netadapter
    gminiports         Same as !ndiskd.netadapter
    miniports          "Classic" version of !ndiskd.netadapter
    filterdb           Same as !ndiskd.netadapter  -filterdb
    offload            Same as !ndiskd.netadapter  -offloads
    ports              Same as !ndiskd.netadapter  -ports
    rcvqueues          Same as !ndiskd.netadapter  -rcvqueues
    filters            Same as !ndiskd.filter
    opens              Same as !ndiskd.mopen
    protocols          Same as !ndiskd.protocol
    nblpools           Same as !ndiskd.nblpool
    nbpools            Same as !ndiskd.nbpool
```

See also

Network Driver Design Guide

Windows Vista and Later Networking Reference

Debugging the Network Stack

NDIS extensions (Ndiskd.dll)

!ndiskd.netadapter

10/23/2019 • 4 minutes to read • Edit Online

The **!ndiskd.netadapter** extension displays information about NDIS miniports, or network adapters, that are active on the system. If you run this command with no parameters, !ndiskd will display a list of all network adapters.

```
!ndiskd.netadapter [-handle <x>] [-basic] [-diag] [-state] [-bindings]
   [-ports] [-offloads] [-filterdb] [-timers] [-rst]
   [-pm] [-ss] [-aoac] [-wol] [-protocoloffloads]
   [-rss] [-hw] [-device] [-wmi] [-custommwmi]
   [-ndiswmi] [-ref] [-log] [-grovel] [-findname <any>]
   [-rcvfilter] [-nicswitch] [-rcvqueues] [-nicswitches] [-iov]
   [-vfs] [-vports] [-iftrace] [-ip]
```

Parameters

-handle
Handle of an NDIS miniport.

-basic
Displays summary information about the miniport.

-diag
Displays auto-diagnostic alerts (if any).

-state
Displays the miniport's current state.

-bindings
Displays miniport bindings.

-ports
Shows a list of NDIS ports.

-offloads
Shows task offload state and capabilities.

-filterdb
Shows the current packet filter.

-timers
Shows timer objects allocated by the miniport.

-rst
Shows Receive-Side Throttling state.

-pm
Shows power management state and capabilities.

-ss
Shows Selective Suspend state.

-aoac

Shows AOAC (Connected Standby) state.

-wol
Shows Wake-on-LAN (WoL) configuration.

-protocoloffloads
Shows active power management protocol offloads.

-rss
Shows Receive Side Scaling parameters.

-hw
Displays hardware resources.

-device
Shows information about the underlying NT device object.

-wmi
Shows WMI GUIDs registered to the adapter.

-customwmi
Shows custom WMI GUIDs registered by the miniport.

-ndiswmi
Shows NDIS-provided WMI GUIDs.

-ref
Shows a breakdown of references on the miniport.

-log
Displays a PnP and Power event log.

-grovel
Forces a search for miniport blocks in memory.

-findname
Filters miniports by name prefix.

-rcvfilter
Shows receive filtering capabilities.

-nicswitch
Shows NIC switch capabilities.

-rcvqueues
Shows receive queues.

-nicswitches
Shows NIC switches.

-iov
Shows SR-IOV (Single Root I/O Virtualization) capabilities.

-vfs
Shows SR-IOV VFs (Virtual Filters).

-vports
Shows Vports (Virtual ports).

-ifrtrace
Shows the in-flight recorder's trace.

-ip
Shows IP addresses on the network's interface.

DLL

Ndiskd.dll

Examples

By running **!ndiskd.netadapter** with no parameters, you can get a list of all network adapters on the system along with their associated miniport drivers. In this example output, look for the Microsoft Kernel Debug Network Adapter, whose handle is ffffdf80140c71a0. For more information about what the Kernel Debug Network Adapter is, see Kernel debugging over the network on the NDIS blog.

```
3: kd> !ndiskd.netadapter
    Driver             NetAdapter          Name
    ffffdf8015a98380   ffffdf8015aa11a0    Microsoft ISATAP Adapter #2
    ffffdf801418d650   ffffdf80140c71a0    Microsoft Kernel Debug Network Adapter
```

By clicking on the handle for the miniport driver or entering the **!ndiskd.netadapter -handle**, you can now see all of NDIS's state on that device. This can be helpful very as a starting place for troubleshooting a network driver or for figuring out where an issue is in the network stack. For example, you can see the Datapath state for the driver and see whether it is connected or not.

At the bottom of the report for this net adapter, there are many other links you can click on to explore further information, such as any pending OIDs and the state of task offloads. These links correspond to many of the parameters for **!ndiskd.netadapter**.

```
3: kd> !ndiskd.netadapter ffffdf80140c71a0


MINIPORT

    Microsoft Kernel Debug Network Adapter

    Ndis handle        ffffdf80140c71a0
    Ndis API version   v6.20
    Adapter context    ffffdf80147d7230
    Driver             ffffdf801418d650 - kdnic  v4.2
    Network interface  ffffdf80139b3a20

    Media type         802.3
    Physical medium    NdisPhysicalMediumOther
    Device instance    ROOT\KDNIC\0000
    Device object      ffffdf80140c7050    More information
    MAC address        18-03-73-c1-e8-72


STATE

    Miniport           Running
    Device PnP         Started             Show state history
    Datapath           Normal
    Interface          Up
    Media              Connected
    Power              D0
    References         0n10                Show detail
    Total resets       0
    Pending OID        None
    Flags              NOT_BUS_MASTER, ALLOW_BUGCHECK_CALLBACK,
                       BUGCHECK_CALLBACK_REGISTERED, DEFAULT_PORT_ACTIVATED,
                       SUPPORTS_MEDIA_SENSE, DOES_NOT_DO_LOOPBACK,
                       MEDIA_CONNECTED
```

```
                        MEDIA_CONNECTED
    PnP flags           VIRTUAL_DEVICE, HIDDEN, NO_HALT_ON_SUSPEND,
                        RECEIVED_START


BINDINGS

    Protocol list       Driver              Open                Context
    MSLLDP              ffffdf80120a5c10     ffffdf8015a749c0    ffffdf8015d325e0
    TCPIP               ffffdf80131cc010     ffffdf801494a650    ffffdf801494aa50
    NDISUIO             ffffdf8015a58140     ffffdf8015a78c10    ffffdf8015a77e00
    TCPIP6              ffffdf80131c9c10     ffffdf80147875a0    ffffdf801494f010
    (RASPPPOE)          Not running
    RSPNDR              ffffdf80120a0c10     ffffdf8015a79c10    ffffdf8015a79010
    LLTDIO              ffffdf8015a5f9b0     ffffdf801406f010    ffffdf8015a786c0
    (RDMANDK)           ffffdf801406d8f0     Declined with NDIS_STATUS_NOT_RECOGNIZED

    Filter list         Driver              Module              Context
    WFP 802.3 MAC Layer LightWeight Filter-0000
                        ffffdf80139a5a70     ffffdf801494c670    ffffdf801494a010
    QoS Packet Scheduler-0000
                        ffffdf8014039d90     ffffdf801494dc70    ffffdf80147dc2b0
    WFP Native MAC Layer LightWeight Filter-0000
                        ffffdf80139fcd70     ffffdf8014950c70    ffffdf8014950880



MORE INFORMATION

    Driver handlers                         Task offloads
    Power management                        PM protocol offloads
    Pending OIDs                            Timers
    Pending NBLs                            Receive side throttling
    Wake-on-LAN (WoL)                       Packet filter
    Receive queues                          Receive filtering
    RSS                                     NIC switch
    Hardware resources                      Selective suspend
    NDIS ports                              WMI guids
    Diagnostic log
```

As example of using **!ndiskd.netadapter** as a starting place for further debugging, click on the "Driver handlers" link at the bottom of the report to see a list of all registered driver callback handlers for this net adapter's miniport driver. In the following example, clicking the link causes !ndiskd to run the **!ndiskd.minidriver** extension with the handle of this net adapter's miniport driver. The miniport driver is the kdnic 4.2 and its handle is ffffdf801418d650.

```
3: kd> !ndiskd.minidriver ffffdf801418d650 -handlers


HANDLERS

    NDIS Handler                        Function pointer   Symbol (if available)
    InitializeHandlerEx                 fffff80f1fd78230  bp
    SetOptionsHandler                   fffff80f1fd72800  bp
    HaltHandlerEx                       fffff80f1fd78040  bp
    ShutdownHandlerEx                   fffff80f1fd722c0  bp

    CheckForHangHandlerEx               fffff80f1fd72810  bp
    ResetHandlerEx                      fffff80f1fd72f70  bp

    PauseHandler                        fffff80f1fd78000  bp
    RestartHandler                      fffff80f1fd78940  bp

    OidRequestHandler                   fffff80f1fd71c90  bp
    CancelOidRequestHandler             fffff80f1fd722c0  bp
    DirectOidRequestHandler             [None]
    CancelDirectOidRequestHandler       [None]
    DevicePnPEventNotifyHandler         fffff80f1fd789a0  bp

    SendNetBufferListsHandler           fffff80f1fd71870  bp
    ReturnNetBufferListsHandler         fffff80f1fd71b50  bp
    CancelSendHandler                   fffff80f1fd722c0  bp
```

You can now click the "bp" link to the right of each handler to set a breakpoint on that handler to debug a particular problem. For example, if there is a hang in the datapath, you can investigate the driver's SendNetBufferListsHandler or ReturnNetBufferListsHandler.

See also

Network Driver Design Guide

Windows Vista and Later Networking Reference

Debugging the Network Stack

NDIS extensions (Ndiskd.dll)

!ndiskd.help

Kernel debugging over the network

!ndiskd.minidriver

!ndiskd.minidriver

10/23/2019 • 2 minutes to read • Edit Online

The **!ndiskd.minidriver** command displays information about an NDIS miniport driver. If you run this extension with no parameters, !ndiskd will display a list of NDIS miniport drivers that are active on the system.

```
!ndiskd.minidriver [-handle <x>] [-basic] [-miniports] [-devices] [-handlers]
```

Parameters

-handle
Handle of an NDIS miniport driver.

-basic
Displays basic information about the miniport driver.

-miniports
Displays the miniports associated with this miniport driver.

-devices
Displays devices associated with this miniport driver.

-handlers
Displays this driver's miniport handlers.

DLL

Ndiskd.dll

Examples

Enter the **!ndiskd.minidriver** command with no parameters to get a list of all NDIS miniport drivers active on the system. In the following example, look for the kdnic adapter's handle, ffffd20d12dec020

```
1: kd> !ndiskd.minidriver -basic
    ffffd20d173deae0 - tunnel
    ffffd20d12dec020 - kdnic
```

Using the handle for the kdnic adapter, you can now click on the handle or enter the **!ndiskd.minidriver -handle** command to see detailed information for the tunnel miniport driver, as well as a list of miniports associated with it.

```
1: kd> !ndiskd.minidriver ffffd20d12dec020


MINIPORT DRIVER

    kdnic

    Ndis handle        ffffd20d12dec020    [type it]
    Driver context     fffff80d2fa15100
    DRIVER_OBJECT      ffffd20d12dee540
    Driver image       kdnic.sys
    Registry path      \REGISTRY\MACHINE\SYSTEM\ControlSet001\Services\kdnic
    Reference Count    2
    Flags              [No flags set]


MINIPORTS

    Miniport
    ffffd20d12dd71a0 - Microsoft Kernel Debug Network Adapter

    Handlers
    Device objects
```

See also

Network Driver Design Guide

Windows Vista and Later Networking Reference

Debugging the Network Stack

NDIS extensions (Ndiskd.dll)

!ndiskd.help

!ndiskd.rcvqueue

10/23/2019 • 2 minutes to read • Edit Online

The **!ndiskd.rcvqueue** command displays information about a receive queue.

```
!ndiskd.rcvqueue [-handle <x>] [-filters] [-mem] [-verbose] [-rcvqueueverbosity <x>]
```

Parameters

-handle
Required. Handle of a receive queue.

-filters
Shows filters on the queue.

-mem
Shows shared memory allocations.

-verbose
Shows additional details.

-rcvqueueverbosity
Level of detail to display.

DLL

Ndiskd.dll

Examples

To obtain the receive queue handle, first enter the **!ndiskd.netadapter** command with no parameters to see the list of net adapters, their drivers, and their handles. In the following example, look for the Microsoft ISATAP Adapter #2's NetAdapter handle, ffff8083e02ce1a0.

```
3: kd> !ndiskd.netadapter
    Driver             NetAdapter          Name
    ffff8083e2668970   ffff8083e02ce1a0    Microsoft ISATAP Adapter #2
    ffff8083e210fae0   ffff8083e0f501a0    Microsoft Kernel Debug Network Adapter
```

Next, with the net adapter's handle, use the **!ndiskd.netadapter -handle -rcvqueues** command to obtain a list of receive queues for this net adapter along with their handles. In this example, there is only one receive queue (the default one) with a handle of ffff8083e3a3d3a0.

```
3: kd> !ndiskd.netadapter ffff8083e02ce1a0 -rcvqueues


RECEIVE QUEUES

    QueueId            Queue Handle        Processor Affinity
    0 [Default]        ffff8083e3a3d3a0    0:0000000000000000 (group:mask)
                       Queue Name:         [Zero-length string]
                       VM Name:            [Zero-length string]
```

Now you can use the queue handle to examine the receive queue details with the **!ndiskd.rcvqueue** command.

```
3: kd> !ndiskd.rcvqueue ffff8083e3a3d3a0


RECEIVE QUEUE

    [Zero-length string]

    VM name            [Zero-length string]
    QueueId            0
    Ndis handle        ffff8083e3a3d3a0

    Miniport           ffff8083e02ce1a0 - Microsoft ISATAP Adapter #2
    Open               [No associated Open]

    Type               Unspecified
    Flags              [No flags set]
    Allocated          Yes
    References         1

    Num filters        0
    Num buffers hint   0
    MSI-X entry        0
    Lookahead size     0
    Processor affinity 0:0000000000000000 (group:mask)

    Receive filter list
    Shared memory allocations
```

See also

Network Driver Design Guide

Windows Vista and Later Networking Reference

Debugging the Network Stack

NDIS extensions (Ndiskd.dll)

!ndiskd.help

!ndiskd.netadapter

!ndiskd.protocol

10/23/2019 • 2 minutes to read • Edit Online

The **!ndiskd.protocol** command displays information about an NDIS protocol driver. If you run this extension with no parameters, !ndiskd will display a list of NDIS protocol drivers that are active on the system.

```
!ndiskd.protocol [-handle <x>] [-findname <any>]
```

Parameters

-handle
Handle of an NDIS protocol.

-findname
Filters protocols by name prefix.

DLL

Ndiskd.dll

Examples

Enter the **!ndiskd.protocol** command to see a list of all NDIS protocols, their handles, and open bindings to miniports (if any). In the following example, look for the TCPIP6TUNNEL protocol's handle, ffff8083e1a95c00.

```
3: kd> !ndiskd.protocol
ffff8083e0114730 - NDISUIO
  ffff8083e55f3010 - Microsoft Kernel Debug Network Adapter

ffff8083e3e90c10 - MSLLDP
  ffff8083e3926010 - Microsoft Kernel Debug Network Adapter

ffff8083e3e98c10 - WANARPV6

ffff8083e3e97010 - WANARP

ffff8083e3e8f6b0 - RSPNDR
  ffff8083e11902c0 - Microsoft Kernel Debug Network Adapter

ffff8083e3e90800 - LLTDIO
  ffff8083e15537d0 - Microsoft Kernel Debug Network Adapter

ffff8083e1a9ac10 - RDMANDK

ffff8083e1a95c00 - TCPIP6TUNNEL
  ffff8083e56b8110 - Microsoft ISATAP Adapter #2

ffff8083e19bec10 - TCPIPTUNNEL

ffff8083e19bfc10 - TCPIP6
  ffff8083e504c770 - Microsoft Kernel Debug Network Adapter

ffff8083e11cec10 - TCPIP
  ffff8083e0c565a0 - Microsoft Kernel Debug Network Adapter
```

With the protocol's handle, now you can enter either click the handle or enter the **!ndiskd.protocol -handle** command to see information for that protocol, such as the handles for the miniports that are bound to it.

```
3: kd> !ndiskd.protocol ffff8083e1a95c00


PROTOCOL

    TCPIP6TUNNEL

    Ndis handle         ffff8083e1a95c00
    Ndis API version    v6.40
    Driver context      fffff80e2e4f9de0
    Driver version      v0.0
    Reference count     2
    Flags               [No flags set]
    Driver image        [Not available]     Why not?


BINDINGS

    Open                Miniport            Miniport Name
    ffff8083e56b8110    ffff8083e02ce1a0    Microsoft ISATAP Adapter #2


HANDLERS

    Protocol handler                        Function pointer   Symbol (if available)
    BindAdapterHandlerEx                    fffff80e2e3baab0   bp
    UnbindAdapterHandlerEx                  fffff80e2e3c1c80   bp
    OpenAdapterCompleteHandlerEx            fffff80e2e4bc940   bp
    CloseAdapterCompleteHandlerEx           fffff80e2e3d19b0   bp
    NetPnPEventHandler                      fffff80e2e3bb140   bp
    UninstallHandler                        [None]
    SendNetBufferListsCompleteHandler       fffff80e2e3919a0   bp
    ReceiveNetBufferListsHandler            fffff80e2e3918a0   bp
    StatusHandlerEx                         fffff80e2e3a9550   bp
    OidRequestCompleteHandler               fffff80e2e398120   bp
    DirectOidRequestCompleteHandler         fffff80e2e398120   bp
```

See also

Network Driver Design Guide

Windows Vista and Later Networking Reference

Debugging the Network Stack

NDIS extensions (Ndiskd.dll)

!ndiskd.help

!ndiskd.mopen

10/23/2019 • 2 minutes to read • Edit Online

The **!ndiskd.mopen** extension displays information about bindings between miniports and protocols. If you run this extension with no parameters, !ndiskd will display a list of all open bindings between NDIS miniport drivers and protocol drivers.

```
!ndiskd.mopen [-handle <x>] [-ref]
```

Parameters

-handle
Handle of an NDIS open binding.

-ref
Shows refcounts of open bindings.

DLL

Ndiskd.dll

Examples

Enter the !ndiskd.mopen command to get a list of all open bindings. In this example, look for the binding between the Microsoft ISATAP Adapter #2 miniport and the TCPIP6TUNNEL protocol. Its handle is ffff8083e56b8110.

```
3: kd> !ndiskd.mopen
Open ffff8083e56b8110
  Miniport: ffff8083e02ce1a0 - Microsoft ISATAP Adapter #2
  Protocol: ffff8083e1a95c00 - TCPIP6TUNNEL

Open ffff8083e11902c0
  Miniport: ffff8083e0f501a0 - Microsoft Kernel Debug Network Adapter
  Protocol: ffff8083e3e8f6b0 - RSPNDR

Open ffff8083e55f3010
  Miniport: ffff8083e0f501a0 - Microsoft Kernel Debug Network Adapter
  Protocol: ffff8083e0114730 - NDISUIO

Open ffff8083e15537d0
  Miniport: ffff8083e0f501a0 - Microsoft Kernel Debug Network Adapter
  Protocol: ffff8083e3e90800 - LLTDIO

Open ffff8083e3926010
  Miniport: ffff8083e0f501a0 - Microsoft Kernel Debug Network Adapter
  Protocol: ffff8083e3e90c10 - MSLLDP

Open ffff8083e0c565a0
  Miniport: ffff8083e0f501a0 - Microsoft Kernel Debug Network Adapter
  Protocol: ffff8083e11cec10 - TCPIP

Open ffff8083e504c770
  Miniport: ffff8083e0f501a0 - Microsoft Kernel Debug Network Adapter
  Protocol: ffff8083e19bfc10 - TCPIP6
```

Now you can either click on the handle or use the handle to enter the **!ndiskd.mopen -handle** command, which

enables you to see more details about that open binding such as its Datapath state and Receive path information.

```
3: kd> !ndiskd.mopen ffff8083e56b8110


OPEN

    Ndis handle        ffff8083e56b8110
    Flags              [No flags set]
    References         1                  Show detail
    Source             1
    Datapath state     Running

    Protocol           ffff8083e1a95c00 - TCPIP6TUNNEL
    Protocol context   ffff8083e15b62e0

    Miniport           ffff8083e02ce1a0 - Microsoft ISATAP Adapter #2
    Miniport context   ffff8083e0772010


RECEIVE PATH

    Packet filter      [No flags set]
    Frame Type(s)      0x86dd
```

See also

Network Driver Design Guide

Windows Vista and Later Networking Reference

Debugging the Network Stack

NDIS extensions (Ndiskd.dll)

!ndiskd.help

!ndiskd.filter

10/23/2019 • 2 minutes to read • Edit Online

The **!ndiskd.filter** extension displays information about an NDIS light-weight filter (LWF). If you run this extension with no parameters, !ndiskd will display a list of all LWFs.

```
!ndiskd.filter [-handle <x>] [-findname <any>] [-handlers]
```

Parameters

-handle
Handle of an NDIS light-weight filter.

-findname
Filters LWFs by name prefix.

-handlers
Displays this LWF's filter handlers.

DLL

Ndiskd.dll

Examples

Enter the **!ndiskd.filter** command with no parameters to get a list of all filters. In this example, look for the ffff8083e14e8460 handle. Note that this handle is for the filter itself and is nested under its associated filter *driver*, the QoS Packet Scheduler.

```
3: kd> !ndiskd.filter
ffff8083e1a7fd90 - QoS Packet Scheduler
  Filter ffff8083e14e8460, Miniport ffff8083e0f501a0 - Microsoft Kernel Debug Network Adapter
ffff8083e1a96b80 - Virtual WiFi Filter Driver
ffff8083e19c4b70 - WFP vSwitch Layers LightWeight Filter
ffff8083e19a6ad0 - WFP Native MAC Layer LightWeight Filter
  Filter ffff8083e43df8f0, Miniport ffff8083e0f501a0 - Microsoft Kernel Debug Network Adapter
ffff8083e19a6d70 - WFP 802.3 MAC Layer LightWeight Filter
  Filter ffff8083e0d89c70, Miniport ffff8083e0f501a0 - Microsoft Kernel Debug Network Adapter
```

By using this filter handle we can now see more detailed information about it, such as it State, Higher filter handle, and Lower filter handle.

```
3: kd> !ndiskd.filter ffff8083e14e8460


FILTER

    Microsoft Kernel Debug Network Adapter-QoS Packet Scheduler-0000

    Ndis handle        ffff8083e14e8460
    Filter driver      ffff8083e1a7fd90 - QoS Packet Scheduler
    Module context     ffff8083e26953e0
    Miniport           ffff8083e0f501a0 - Microsoft Kernel Debug Network Adapter
    Network interface  ffff8083e200f010

    State              Running
    Datapath           Send only
    References         1
    Flags              RUNNING
    More flags         OID_TOP

    Higher filter      ffff8083e0d89c70 - Microsoft Kernel Debug Network Adapter-WFP 802.3 MAC Layer
LightWeight Filter-0000
    Lower filter       ffff8083e43df8f0 - Microsoft Kernel Debug Network Adapter-WFP Native MAC Layer
LightWeight Filter-0000

    Driver handlers
```

See also

Network Driver Design Guide

Windows Vista and Later Networking Reference

Debugging the Network Stack

NDIS extensions (Ndiskd.dll)

!ndiskd.help

!ndiskd.filterdriver

10/23/2019 • 2 minutes to read • Edit Online

The **!ndiskd.filterdriver** extension displays information about an NDIS filter driver. If you run this extension with no parameters, !ndiskd will display a list of all filter drivers.

```
!ndiskd.filterdriver [-handle <x>] [-filters] [-handlers]
```

Parameters

-handle
Handle of an NDIS filter driver.

-filters
Displays instances of this driver's filters.

-handlers
Displays this driver's filter handlers.

DLL

Ndiskd.dll

Examples

Run **!ndiskd.filterdriver** with no parameters to see a list of all filter drivers on the system. In the following example, look for the Virtual WiFi Filter Driver, whose handle is ffffbc064cc83be0.

```
0: kd> !ndiskd.filterdriver
    ffffbc064ccd4900 - QoS Packet Scheduler
    ffffbc064cc83be0 - Virtual WiFi Filter Driver
    ffffbc064cb91a10 - WFP vSwitch Layers LightWeight Filter
    ffffbc064cb8fd70 - WFP Native MAC Layer LightWeight Filter
    ffffbc064cb59b00 - WFP 802.3 MAC Layer LightWeight Filter
```

By clicking the filter driver handle from the previous example or by using it to enter the **!ndiskd.filterdriver -handle** command in the command window, you can get see more detailed information about that filter driver. In this case, for example, there are no filter modules for this filter driver.

```
0: kd> !ndiskd.filterdriver ffffbc064cc83be0


FILTER DRIVER

    Virtual WiFi Filter Driver

    Ndis handle        ffffbc064cc83be0
    Driver context     ffffbc064cc8e9d0
    Ndis API version   v6.50
    Driver version     v1.0
    Driver object      ffffbc064cc8e9d0
    Driver image       vwififlt.sys

    Bind flags         Optional, Modifying
    Class              [Zero-length string]
    References         1


FILTER MODULES

    Filter module
    [No filter modules were found]


HANDLERS

    Filter handler                      Function pointer   Symbol (if available)
    SetOptionsHandler                   [None]
    SetFilterModuleOptionsHandler       [None]
    AttachHandler                       fffff80787d83b60  bp
    DetachHandler                       fffff80787d84800  bp
    RestartHandler                      fffff80787d86e20  bp
    PauseHandler                        fffff80787d863e0  bp
    SendNetBufferListsHandler           fffff80787d814d0  bp
    SendNetBufferListsCompleteHandler   fffff80787d81940  bp
    CancelSendNetBufferListsHandler     fffff80787d842f0  bp
    ReceiveNetBufferListsHandler        fffff80787d817e0  bp
    ReturnNetBufferListsHandler         fffff80787d81a80  bp
    OidRequestHandler                   fffff80787d85ae0  bp
    OidRequestCompleteHandler           fffff80787d85fd0  bp
    DirectOidRequestHandler             fffff80787d84af0  bp
    DirectOidRequestCompleteHandler     fffff80787d84e80  bp
    CancelDirectOidRequestHandler       fffff80787d841d0  bp
    DevicePnPEventNotifyHandler         fffff80787d849e0  bp
    NetPnPEventHandler                  fffff80787d85a40  bp
    StatusHandler                       fffff80787d877c0  bp
```

See also

Network Driver Design Guide

Windows Vista and Later Networking Reference

Debugging the Network Stack

NDIS extensions (Ndiskd.dll)

!ndiskd.help

!ndiskd.nbl

10/23/2019 • 2 minutes to read • Edit Online

The **!ndiskd.nbl** extension displays information about a **NET_BUFFER_LIST** (NBL) structure.

```
!ndiskd.nbl [-handle <x>] [-basic] [-chain] [-info] [-data]
[-netmon] [-capfile <str>] [-launch] [-overwrite] [-log]
[-stacks] [-NblCurrentOwner]
```

Parameters

-handle
Required. Address of a **NET_BUFFER_LIST** structure.

-basic
Displays basic information about an NBL.

-chain
Displays all the NBLs and **NET_BUFFER**s in an NBL chain.

-info
Displays all the out-of-band information that is associated with an NBL.

-data
Displays the actual data payload of an NBL.

-netmon
Views the NBL chain in Microsoft Network Monitor.

-capfile
Specifies the path to which a netmon capture is saved.

-launch
Automatically launches netmon.exe after saving the capture file.

-overwrite
Allows overwriting the capture file if it already exists.

-log
Shows NBL log if NBL history logging is enabled.

-stacks
Includes callstacks with NBL log (use with -log).

-NblCurrentOwner
Shows the current owner of the NBL.

DLL

Ndiskd.dll

Examples

In the following example, NBL tracking has been enabled to extract a handle for an NBL from the NBL log. For

more information about NBL tracking and the NBL log, see **!ndiskd.nbllog**.

At the time of log collection, the NBL in this example was returned by the TCPIP6 protocol to the WFP Native Mac Layer LightWeight Filter.

```
2: kd> !ndiskd.nbl ffffdf80149524a0
    NBL               ffffdf80149524a0    Next NBL           NULL
    First NB          ffffdf8014952610    Source             ffffdf80140c71a0 - Microsoft Kernel Debug
Network Adapter
    Flags             INDICATED, RETURNED, NBL_ALLOCATED, PROTOCOL_020_0,
                      PROTOCOL_200_0

    Walk the NBL chain                    Dump data payload
    Show out-of-band information
    Review NBL history
```

By clicking on the "Dump data payload" link from the previous example or by entering the **!ndiskd.nbl -handle -data** command, you can see the data payload of this NBL. In the following example, the NBL contains only one **NET_BUFFER** structure. To further explore the contents of that **NET_BUFFER** structure, run the **!ndiskd.nb -handle** command with its handle.

```
2: kd> !ndiskd.nbl ffffdf80149524a0 -data
NET_BUFFER ffffdf8014952610
```

See also

Network Driver Design Guide

Windows Vista and Later Networking Reference

Debugging the Network Stack

NDIS extensions (Ndiskd.dll)

!ndiskd.help

NET_BUFFER_LIST

NET_BUFFER

!ndiskd.nbllog

!ndiskd.nb

!ndiskd.nb

10/23/2019 • 2 minutes to read • Edit Online

The **!ndiskd.nb** extension displays information about a **NET_BUFFER** (NB) structure.

```
!ndiskd.nb [-handle <x>] [-verbosity <x>] [-basic] [-chain] [-data]
```

Parameters

-handle
Required. Address of a **NET_BUFFER** structure.

-verbosity
Level of detail to display.

-basic
Displays basic information about an NB.

-chain
Displays all the MDLs associated with an NB.

-data
Dumps the actual data payload of an NB.

DLL

Ndiskd.dll

Examples

The **NET_BUFFER** in the following examples was obtained from the **NET_BUFFER_LIST** in the Examples section of the **!ndiskd.nbl** topic. The NB's handle is ffffdf8014952610.

```
2: kd> !ndiskd.nbl ffffdf80149524a0 -data
NET_BUFFER ffffdf8014952610
```

You can click the **NET_BUFFER**'s handle or run the **!ndiskd.nb -handle** command to see its details.

```
2: kd> !ndiskd.nb ffffdf8014952610
    NB                ffffdf8014952610    Next NB            0
    Length            0                   Source pool        ffffdf80147e4a40
    First MDL         ffffdf8014a37930    DataOffset         0
    Current MDL       [First MDL]         Current MDL offset 0

    View associated NBL
```

Use the **!ndiskd.nb -chain** command to see this **NET_BUFFER**'s MDL chain in addition to its basic details. In the following example, there is only one MDL. Its handle is ffffdf8014a37930.

```
2: kd> !ndiskd.nb ffffdf8014952610 -chain
    NB                ffffdf8014952610    Next NB            0
    Length            0                   Source pool        ffffdf80147e4a40
    First MDL         ffffdf8014a37930    DataOffset         0
    Current MDL       [First MDL]         Current MDL offset 0
        MDL [current]     ffffdf8014a37930    MDL Flags             c
        MappedSystemVa    ffffdf8014bf0024    ByteCount          0n1514
        Process           [System process]    ByteOffset         0n36
```

See also

Network Driver Design Guide

Windows Vista and Later Networking Reference

Debugging the Network Stack

NDIS extensions (Ndiskd.dll)

!ndiskd.help

NET_BUFFER

NET_BUFFER_LIST

!ndiskd.nbl

!ndiskd.nblpool

10/23/2019 • 2 minutes to read • Edit Online

The **!ndiskd.nblpool** extension displays information about a **NET_BUFFER_LIST** (NBL) pool. If you run this extension with no parameters, !ndiskd will display a list of all allocated NBL pools in the system.

```
!ndiskd.nblpool [-handle <x>] [-basic] [-allocations] [-find <str>] [-findnb <str>]
    [-findctx <str>] [-findctxtype <str>] [-findva <x>] [-findpa <x>]
```

Parameters

-handle
Handle of an NBL pool.

-basic
Displays basic information about the NBL pool.

-allocations
Displays all allocated NBLs.

-find
Filter the list of allocated NBLs using a debugger expression.

-findnb
Filter the list of allocated NBLs by linked **NET_BUFFER**s (NBs).

-findctx
Filter the list of allocated NBLs by context area.

-findctxtype
Override the datatype of the context area.

-findva
Find NBLs that contain an NB that straddles the given virtual address.

-findpa
Find NBLs that contain an NB that straddles the given physical address.

DLL

Ndiskd.dll

Examples

Enter the **!ndiskd.nblpool** command with no parameters to see a list of all allocated NBL pools. In this example, look for the NBL pool allocated by the kernel debugger network interface card (kdnic) with the KDNr Tag. Its handle is ffffdf80147e4a40.

```
2: kd> !ndiskd.nblpool
    NBL Pool           Tag                Allocated by
    ffffdf80179b6a40   NiBP               WdNisDrv!CWFPLayer::Initialize+c6
    ffffdf8015ac6a40   EUNP               tunnel!TunnelEtherUdpGlobalInit+81
    ffffdf8015a78040   Nuio               ndisuio!ndisuioCreateBinding+15f
    ffffdf8015a77800   Nuio               ndisuio!ndisuioCreateBinding+13c
    ffffdf8015a63040   BaNB               rspndr!TopStartNetBufferModule+6d
    ffffdf8015a68a40   LLnb               mslldp!lldpProtSetOptions+49
    ffffdf8014654040   BaNB               lltdio!TopStartNetBufferModule+6d
    ffffdf801494ca40   Pcsb               pacer!PcFilterAttach+142
    ffffdf80147e4a40   KDNr               kdnic!NICAllocAdapter+178
    ffffdf80131ce040   bnvW               wfplwfs!DriverEntry+7a0
    ffffdf80139ffa40   Wfdp               wfplwfs!WfpRioInitialize+a4
    ffffdf8012061200   UNbl               NETIO!NetioAllocateNetBufferListNetBufferMdlAndDataPool+49
    ffffdf8013968a40   TcDN               NETIO!NetioAllocateNetBufferListNetBufferMdlAndDataPool+49
    ffffdf8013969a40   TNbl               NETIO!NetioAllocateNetBufferListNetBufferMdlAndDataPool+49
    ffffdf801397c040   StBn               NETIO!StreamPoolsInit+c1
    ffffdf8013088040   Wfra               NETIO!WfpNblInfoLibraryInit+b8
    ffffdf8012067440   Nnnn               NETIO!NetioInitializeNetBufferListLibrary+13e
    ffffdf8012067a40   Nnbl               NETIO!NetioInitializeNetBufferListLibrary+112
    ffffdf80131caa40   NDrt               ndis!ndisInitializePeriodicReceives+22f
    ffffdf80131d5a40   NDnd               ndis!DriverEntry+5e9
```

Click on the NBL pool's handle or enter the **!ndiskd.nblpool -handle** command to examine its details.

```
2: kd> !ndiskd.nblpool ffffdf80147e4a40


NBL POOL

    Ndis handle        ffffdf80147e4a40
    Allocation tag     KDNr
    Owner
    Allocated by       kdnic!NICAllocAdapter+178

    Flags              CONTAINS_NET_BUFFER
    Structure size     0n544
    Context size       0
    Data size          0

    All allocated NBLs
```

To explore the NBLs contained in this NBL pool, click on the "All allocated NBLs" link at the bottom. Alternatively, you can also enter the **!ndiskd.nblpool -handle -allocations** command. As shown in the following example, this NBL pool contains more than 1024 NBLs so !ndiskd quit early. You can use the -force option to work around this limit and see all of the NBLs in this NBL pool.

```
2: kd> !ndiskd.nblpool ffffdf80147e4a40 -allocations


ALL ALLOCATED NBLs

    NBL                 Active?
    ffffdf8014951940    Allocated
    ffffdf8014951b90    Allocated
    ffffdf8014951de0    Allocated
    ffffdf8014951030    Allocated
    ffffdf80149524a0    Allocated
    ffffdf80149526f0    Allocated
    ffffdf8014952940    Allocated
    ffffdf8014952b90    Allocated
    ffffdf8014952de0    Allocated
    ffffdf8014952030    Allocated
    ffffdf80149534a0    Allocated
    ffffdf80149536f0    Allocated
    ffffdf8014953940    Allocated
    ffffdf8014953b90    Allocated
    ffffdf8014953de0    Allocated
    ffffdf8014953030    Allocated
    ffffdf80149544a0    Allocated
    ffffdf80149546f0    Allocated
    ffffdf8014954940    Allocated

...

    ffffdf80148b0b90    Allocated
    ffffdf80148b0de0    Allocated
    ffffdf80148b0030    Allocated
    ffffdf80148b14a0    Allocated
    ffffdf80148b16f0    Allocated
    ffffdf80148b1940    Allocated
    ffffdf80148b1b90    Allocated
    ffffdf80148b1de0    Allocated
    ffffdf80148b1030    Allocated
    [Maximum of 1024 items read; quitting early. Rerun with the '-force' option
    to bypass this limit.]
```

See also

Network Driver Design Guide

Windows Vista and Later Networking Reference

Debugging the Network Stack

NDIS extensions (Ndiskd.dll)

!ndiskd.help

NET_BUFFER_LIST

NET_BUFFER

!ndiskd.nbpool

10/23/2019 • 2 minutes to read • Edit Online

The **!ndiskd.nbpool** extension displays information about a **NET_BUFFER** (NB) pool. If you run this extension with no parameters, !ndiskd will display a list of all allocated NB pools in the system.

```
!ndiskd.nbpool [-handle <x>] [-allocations] [-find <str>] [-findva <x>] [-findpa <x>]
```

Parameters

-handle
Handle of an NB pool.

-allocations
Displays all allocated NBs.

-find
Filter the list of allocated NBs using a debugger expression.

-findva
Find NBs that straddle the given virtual address.

-findpa
Find NBs that straddle the given physical address.

DLL

Ndiskd.dll

Examples

Enter the **!ndiskd.nbpool** command with no parameters to see a list of all allocated NB pools. In this example, look for the NB pool allocated by the Netio service with the Nnbf Tag. Its handle is ffffdf801308ca40.

```
2: kd> !ndiskd.nbpool
    NB Pool            Tag                Allocated by
    ffffdf8013963a40   UDNb               NETIO!NetioAllocateNetBufferMdlAndDataPool+3c
    ffffdf801396aa40   TSNb               NETIO!NetioAllocateNetBufferMdlAndDataPool+3c
    ffffdf801397d4c0   StBn               NETIO!StreamPoolsInit+90
    ffffdf801308ca40   Nnbf               NETIO!NetioInitializeNetBufferListLibrary+dd
    ffffdf80131cba40   NDnd               ndis!DriverEntry+615
```

Click on the NB pool's handle or enter the **!ndiskd.nbpool -handle** command to examine its details.

```
2: kd> !ndiskd.nbpool ffffdf801308ca40


NB POOL

    Ndis handle        ffffdf801308ca40
    Allocation tag     Nnbf
    Owner
    Allocated by       NETIO!NetioInitializeNetBufferListLibrary+dd

    Flags              [No flags set]
    Structure size     0n176
    Data size          0

    All allocated NBs
```

To explore the NBs contained in this NB pool, click on the "All allocated NBs" link at the bottom. Alternatively, you can also enter the **!ndiskd.nbpool -handle -allocations** command. As shown in the following example, this NB pool contains more than 1024 NBs so !ndiskd quit early. You can use the -force option to work around this limit and see all of the NBs in this NB pool.

```
2: kd> !ndiskd.nbpool ffffdf801308ca40 -allocations


ALL ALLOCATED NBs

    NB                 Active?
    ffffdf8016ea4360   Allocated
    ffffdf801744df50   Allocated
    ffffdf8016932860   Allocated
    ffffdf8016e31500   Allocated
    ffffdf80174eade0   Allocated
    ffffdf8017daa900   Allocated
    ffffdf8017c8c680   Allocated
    ffffdf80166b23b0   Allocated
    ffffdf80164fea70   Allocated
    ffffdf8012845990   Allocated
    ffffdf8017d692d0   Allocated
    ffffdf8017cdc090   Allocated
    ffffdf8012771780   Allocated
    ffffdf80158a3550   Allocated
    ffffdf8012eef5c0   Allocated
    ffffdf80127719d0   Allocated
    ffffdf8015119570   Allocated
    ffffdf8012e18d40   Allocated
    ffffdf8017929b10   Allocated
    ffffdf8016d4e430   Allocated

...

    ffffdf8015ffbbd0   Allocated
    ffffdf8015ec1b10   Freed
    ffffdf80158e56d0   Allocated
    ffffdf8016272110   Freed
    ffffdf8015d8e030   Freed
    ffffdf8015d8e770   Freed
    ffffdf80158ddc30   Freed
    ffffdf801584acc0   Freed
    ffffdf8015846b40   Freed
    ffffdf8015a06c50   Freed
    ffffdf801480c300   Freed
    ffffdf8015e48f50   Freed
    ffffdf8015de64e0   Freed
    ffffdf8015ddff50   Freed
    [Maximum of 1024 items read; quitting early. Rerun with the '-force' option
    to bypass this limit.]
```

See also

Network Driver Design Guide

Windows Vista and Later Networking Reference

Debugging the Network Stack

NDIS extensions (Ndiskd.dll)

!ndiskd.help

NET_BUFFER

!ndiskd.pendingnbls

10/23/2019 • 2 minutes to read • Edit Online

The **!ndiskd.pendingnbls** extension displays pending NBLs (**NET_BUFFER_LISTs**) that are in transit.

```
!ndiskd.pendingnbls [-handle <x>] [-fullstack] [-verbosity <x>]
```

Parameters

-handle
Handle of an NDIS miniport, filter, or open.

-fullstack
Shows pending NBLs from the entire stack associated with the handle.

-verbosity
Level of detail to display.

DLL

Ndiskd.dll

Examples

!ndiskd.pendingnbls can be passed the handle of an NDIS miniport, filter, or open. The following series of examples use a miniport handle. To see a list of all miniports and their associated minidrivers, run the **!ndiskd.netadapter** extension with no parameters. In the following example output, look for the Microsoft Kernel Debug Network Adapter, whose handle is ffffe00bc3f701a0. Its minidriver's handle is ffffe00bc51b9ae0.

```
0: kd> !ndiskd.netadapter
    Driver             NetAdapter          Name
    ffffe00bc6e12ae0   ffffe00bc6e4e1a0    Microsoft ISATAP Adapter #2
    ffffe00bc51b9ae0   ffffe00bc3f701a0    Microsoft Kernel Debug Network Adapter
```

To see the pending NBLs for a miniport, set a breakpoint on its minidriver's SendNetBufferListsHandler. Use the minidriver's handle to run the **!ndiskd.minidriver -handle -handlers** command to see a list of its handlers, then click the "bp" link to the right of the SendNetBufferListsHandler. You can alternatively enter the **bp -handle** command in the command line.

```
0: kd> !ndiskd.minidriver ffffe00bc51b9ae0 -handlers


HANDLERS

    NDIS Handler                        Function pointer   Symbol (if available)
    InitializeHandlerEx                 fffff80ae9618230  bp
    SetOptionsHandler                   fffff80ae9612800  bp
    HaltHandlerEx                       fffff80ae9618040  bp
    ShutdownHandlerEx                   fffff80ae96122c0  bp

    CheckForHangHandlerEx               fffff80ae9612810  bp
    ResetHandlerEx                      fffff80ae9612f70  bp

    PauseHandler                        fffff80ae9618000  bp
    RestartHandler                      fffff80ae9618940  bp

    OidRequestHandler                   fffff80ae9611c90  bp
    CancelOidRequestHandler             fffff80ae96122c0  bp
    DirectOidRequestHandler             [None]
    CancelDirectOidRequestHandler       [None]
    DevicePnPEventNotifyHandler         fffff80ae96189a0  bp

    SendNetBufferListsHandler           fffff80ae9611870  bp
    ReturnNetBufferListsHandler         fffff80ae9611b50  bp
    CancelSendHandler                   fffff80ae96122c0  bp
```

After setting the breakpoint on the SendNetBufferListsHandler, enter the **g** command to let the debugee target machine run and hit the breakpoint.

```
0: kd> bp fffff80ae9611870
0: kd> g
Breakpoint 0 hit
fffff80a`e9611870 4053            push    rbx
```

Now, after hitting the minidriver's SendNetBufferListsHandler breakpoint, you can see any pending NBLs for the miniport by entering the **!ndiskd.pendingnbls -handle** command with the miniport's handle.

Note The debugee target machine in this example was loading a web page when it hit the breakpoint, so traffic was flowing through the miniport's datapath. Therefore, it had a pending NBL to send. Even after setting a breakpoint on one or more of the NBL handlers for the minidriver, you may not see any pending NBLs if there were no activity in the datapath.

```
0: kd> !ndiskd.pendingnbls ffffe00bc3f701a0

PHASE 1/3: Found 20 NBL pool(s).
PHASE 2/3: Found 342 freed NBL(s).

    Pending Nbl        Currently held by
    ffffe00bc5545c60   ffffe00bc3f701a0 - Microsoft Kernel Debug Network Adapter  [NetAdapter]


PHASE 3/3: Found 1 pending NBL(s) of 4817 total NBL(s).
Search complete.
```

See also

Network Driver Design Guide

NDIS extensions (Ndiskd.dll)

!ndiskd.help

NET_BUFFER_LIST

!ndiskd.netadapter

!ndiskd.minidriver

bp, bu, bm (Set Breakpoint)

!ndiskd.nbllog

10/23/2019 • 2 minutes to read • Edit Online

The **!ndiskd.nbllog** extension displays the log of all NBL (**NET_BUFFER_LIST**) activity on the system.

```
!ndiskd.nbllog [-stacks]
```

Parameters

-stacks
Include callstacks.

DLL

Ndiskd.dll

Remarks

Important **!ndiskd.nbllog** requires NBL tracking to be enabled on the debugee target machine. NBL tracking is not enabled by default in all configurations of Windows. If NBL tracking is not enabled, !ndiskd will give you instructions on how to enable it, as shown in the following snippet.

```
0: kd> !ndiskd.nbllog
    This command requires NBL tracking to be enabled on the debugee target
    machine.  (By default, client operating systems have level 1, and servers
    have level 0).  To enable, set this REG_DWORD value to a nonzero value on
    the target machine and reboot the target machine:

    HKLM\SYSTEM\CurrentControlSet\Services\NDIS\Parameters ! TrackNblOwner
    Possible Values (features are cumulative)
    * 0:  Disable all tracking.
    * 1:  Track the most recent owner of each NBL (enables !ndiskd.pendingnbls)
    * 2:  Scan for leaks at runtime (use with StuckNblReaction)
    * 3:  Keep a full history of all activity (enables !ndiskd.nbl -log)
    * 4:  Take stack capture snapshots (enables !ndiskd.nbl -log -stacks)
    This command requires level 3 or higher.
```

The NBL log shows network traffic on the system. **!ndiskd.netreport** parses the NBL tracking log to display this network traffic visually. Therefore, if NBL tracking is not enabled, **!ndiskd.netreport** will not be able to show you this information.

Examples

After you have enabled NBL tracking on the target debugee machine, enter the **!ndiskd.nbllog** command to see the log of all NBL traffic on the system. As shown in the example below, running **!ndiskd.nbllog** with no parameters will limit output to 200 events, which can be bypassed by rerunning the command with the *-force* option. The middle of the output in this example has been excised for brevity.

```
0: kd> !ndiskd.nbllog
    NBLs               Processor           Event              Detail

    ffffe00bc71453f0   CPU  0              Freed
    ffffe00bc7163b40   CPU  2              Allocated
    ffffe00bc7163b40   CPU  2              ProtocolSent       ffffe00bc5ac4880 - QoS Packet Scheduler-0000
    ffffe00bc7163b40   CPU  2              FilterSent         ffffe00bc5ac5c70 - WFP Native MAC Layer
LightWeight Filter-0000
    ffffe00bc7163b40   CPU  2, IRQL=DPC    FilterSent         ffffe00bc3f701a0 - Microsoft Kernel Debug
Network Adapter
    ffffe00bc7163b40   CPU  2, IRQL=DPC    SentToMiniport     ffffe00bc3f701a0 - Microsoft Kernel Debug
Network Adapter
    ffffe00bc7163b40   CPU  0, IRQL=DPC    MiniportSendCompleted ffffe00bc5ac5c70 - WFP Native MAC Layer
LightWeight Filter-0000
    ffffe00bc7163b40   CPU  0, IRQL=DPC    FilterSendCompleted ffffe00bc5ac4880 - QoS Packet Scheduler-0000
    ffffe00bc7163b40   CPU  0, IRQL=DPC    FilterSendCompleted send complete in NDIS, sorting to Opens
    ffffe00bc7163b40   CPU  0, IRQL=DPC    SendCompleted      ffffe00bc5ab7c10 - TCPIP6

...

    ffffe00bc6b469b0   CPU  2              Allocated
    ffffe00bc6b469b0   CPU  2              Freed
    ffffe00bc64a3690   CPU  2              Allocated
    ffffe00bc64a3690   CPU  2              ProtocolSent       ffffe00bc5ac4880 - QoS Packet Scheduler-0000
    ffffe00bc64a3690   CPU  2              FilterSent         ffffe00bc5ac5c70 - WFP Native MAC Layer
LightWeight Filter-0000
    ffffe00bc64a3690   CPU  2, IRQL=DPC    FilterSent         ffffe00bc3f701a0 - Microsoft Kernel Debug
Network Adapter
    ffffe00bc64a3690   CPU  2, IRQL=DPC    SentToMiniport     ffffe00bc3f701a0 - Microsoft Kernel Debug
Network Adapter
    ffffe00bc3cf2d10   CPU  1              Allocated
    ffffe00bc7bc6030   CPU  1              Allocated
    ffffe00bc3cf2d10   CPU  1              ProtocolSent       ffffe00bc5ac4880 - QoS Packet Scheduler-0000

    Maximum of 200 events printed; quitting early.
    Rerun with the '-force' option to bypass this limit.
```

For a more detailed description of how to interpret the results of **!ndiskd.nbllog**, see !ndiskd.nbl -log on the NDIS blog.

See also

Network Driver Design Guide

Windows Vista and Later Networking Reference

Debugging the Network Stack

NDIS extensions (Ndiskd.dll)

!ndiskd.help

NET_BUFFER_LIST

!ndiskd.nbl -log

!ndiskd.cxadapter

10/23/2019 • 2 minutes to read • Edit Online

The **!ndiskd.cxadapter** extension displays information about a NETADAPTER object.

For more information about the Network Adapter WDF Class Extension (NetAdapterCx), see Network Adapter WDF Class Extension (Cx).

```
!ndiskd.cxadapter [-handle <x>] [-basic] [-power] [-datapath]
```

Parameters

-handle
Required. Handle of a NETADAPTER.

-basic
Displays basic information.

-power
Displays information about the NETPOWERSETTINGS object of the NETADAPTER.

-datapath
Displays information about the datapath queues.

DLL

Ndiskd.dll

Examples

To obtain a handle for a NETADAPTER object, first run the **!ndiskd.netadapter** command to see a list of all NIC drivers and NetAdapters on the system. In the following example, look for the handle for the NetAdapter named Realtek PCIe GBE Family Controller NetAdapter Sample Driver #2. Its handle is ffffd1022d048030.

```
0: kd> !ndiskd.netadapter
    Driver             NetAdapter          Name
    ffffd1022e8ecae0   ffffd1022d048030    Realtek PCIe GBE Family Controller NetAdapter Sample Driver #2
    ffffd1022ed908e0   ffffd1022e8611a0    Microsoft Kernel Debug Network Adapter
```

By clicking on this NetAdapter's handle or by entering the **!ndiskd.netadapter -handle** command with its handle on the command line, you can see details for this NetAdapter, including its NETADAPTER object. The Realtek PCIe GBE Family Controller NetAdapter Sample Driver #2's NETADAPTER handle is 00002efdd0e5f988.

```
0: kd> !ndiskd.netadapter ffffd1022d048030


NETADAPTER

    Realtek PCIe GBE Family Controller NetAdapter Sample Driver #2

    Ndis handle        ffffd1022d048030
    NETADAPTER         00002efdd0e5f988    More information
    WDFDEVICE          00002efdcf45f2f8
    Driver             ffffd1022e8ecae0 - RtEthSample  v0.0
```

```
    Network interface  ffffd1022e395a20

    Media type         802.3
    Device instance    PCI\VEN_10EC&DEV_8168&SUBSYS_816810EC&REV_03\4&22bb23f1&0&0038
    Device object      ffffd1022de127f0    More information
    MAC address        00-e0-4c-68-00-8b


STATE

    Miniport           Running
    Device PnP         Started             Show state history
    Datapath           Normal
    Interface          Up
    Media              Connected
    Power              D0
    References         0n10                Show detail
    Total resets       0
    Pending OID        None
    Flags              NOT_BUS_MASTER, WDF, DEFAULT_PORT_ACTIVATED,
                       SUPPORTS_MEDIA_SENSE, DOES_NOT_DO_LOOPBACK,
                       MEDIA_CONNECTED
    PnP flags          PM_SUPPORTED, DEVICE_POWER_ENABLED,
                       DEVICE_POWER_WAKE_ENABLE, HARDWARE_DEVICE,
                       NDIS_WDM_DRIVER, WAKE_CAPABLE


IP ADDRESS SUMMARY

    10.137.188.169                         See all IP addresses on this adapter
    fe80::3cad:81bb:5dad:1066


BINDINGS

    Protocol list      Driver              Open               Context
    MSLLDP             ffffd1023043f6a0    ffffd1022e786a90   ffffd102307465c0
    LLTDIO             ffffd1022c6b7830    ffffd1022ef8cc00   ffffd1022f1e5730
    TCPIP6             ffffd1022e2c7c10    ffffd10230b98310   ffffd102304d9010
    (RASPPPOE)         Not running
    (RDMANDK)          ffffd1022d574a70    Declined with NDIS_STATUS_NOT_RECOGNIZED
    RSPNDR             ffffd1022c71a830    ffffd1022de0cc00   ffffd1022d03f6a0
    TCPIP              ffffd1022e2cbc10    ffffd1022de067f0   ffffd1022d03f010
    NDISUIO            ffffd1022de07670    ffffd1022cd648d0   ffffd10231131970

    Filter list        Driver              Module             Context
    WFP 802.3 MAC Layer LightWeight Filter-0000
                       ffffd1022e384d70    ffffd1022f271660   ffffd10230cfe2c0
    QoS Packet Scheduler-0000
                       ffffd1022d56f220    ffffd1022f26d660   ffffd10231778700
    WFP Native MAC Layer LightWeight Filter-0000
                       ffffd1022e384ad0    ffffd1022f26b660   ffffd1022ed59c20



MORE INFORMATION

    Driver handlers                        Task offloads
    Power management                       PM protocol offloads
    Pending OIDs
    Pending NBLs                           Receive side throttling
    Wake-on-LAN (WoL)                      Packet filter
    Receive queues                         Receive filtering
    RSS                                    NIC switch
                                           Selective suspend
    NDIS ports                             WMI guids
    Diagnostic log
```

Because the NETADAPTER object is a WDF object, clicking its handle will cause the debugger to run the **!wdfkd.wdfhandle** command which will give you more information about it from a WDF perspective. To see more detailed information about the NETADAPTER from a networking perspective, click the "More Information" link to the right of the NETADAPTER's handle to run the **!ndiskd.cxadapter** command with its handle.

```
0: kd> !ndiskd.cxadapter ffffd1022f1a0720


NETADAPTER

    Miniport           ffffd1022d048030 - Realtek PCIe GBE Family Controller NetAdapter Sample Driver #2
    NETADAPTER         00002efdd0e5f988
    WDFDEVICE          00002efdcf45f2f8

    Event Callbacks                        Function pointer   Symbol (if available)
    EvtAdapterCreateTxQueue                fffff80034151508   RtEthSample+1508
    EvtAdapterCreateRxQueue                fffff800341510ec   RtEthSample+10ec

    Show datapath info
    Show NETPOWERSETTINGS info
```

You can also combine this command other parameters such as *-datapath* to see more information for this NETADAPTER.

```
0: kd> !ndiskd.cxadapter ffffd1022f1a0720 -basic -datapath


NETADAPTER

    Miniport           ffffd1022d048030 - Realtek PCIe GBE Family Controller NetAdapter Sample Driver #2
    NETADAPTER         00002efdd0e5f988
    WDFDEVICE          00002efdcf45f2f8

    Event Callbacks                        Function pointer   Symbol (if available)
    EvtAdapterCreateTxQueue                fffff80034151508   RtEthSample+1508
    EvtAdapterCreateRxQueue                fffff800341510ec   RtEthSample+10ec


DATAPATH QUEUES

    NETTXQUEUE         ffffd1022f512700
    NETRXQUEUE         ffffd1022cc7b0d0
```

See also

Network Driver Design Guide

Windows Vista and Later Networking Reference

Debugging the Network Stack

NDIS extensions (Ndiskd.dll)

!ndiskd.help

Network Adapter WDF Class Extension (Cx)

!ndiskd.netadapter

!wdfkd.wdfhandle

!ndiskd.netqueue

10/23/2019 • 2 minutes to read • Edit Online

The **!ndiskd.netqueue** extension displays information about a NETTXQUEUE or NETRXQUEUE object.

For more information about the Network Adapter WDF Class Extension (NetAdapterCx), see Network Adapter WDF Class Extension (Cx).

```
!ndiskd.netqueue [-handle <x>] [-basic]
```

Parameters

-handle
Required. Handle of a NETTXQUEUE or NETRXQUEUE.

-basic
Displays basic information.

DLL

Ndiskd.dll

Examples

Note See Summary of Objects to see a diagram explaining the relationship of the NETTXQUEUE and NETRXQUEUE objects with other objects in the NetAdapterCx.

To obtain a handle for a NETTXQUEUE or NETRXQUEUE, follow these steps:

1. Run the **!ndiskd.netadapter** extension.
2. Click on the handle for a NetAdapter that has a NetAdapterCx driver installed.
3. Click the "More Information" link to the right of the NetAdapter's NETADAPTER object to run the **!ndiskd.cxadapter** extension.
4. Enter the **!ndiskd.cxadapter** command with the *-datapath* parameter to see that NETADAPTER's datapath queues.

For details on this procedure, see the examples on the **!ndiskd.cxadapter** topic. In the following example, look for the handle for this NETADAPTER's NETTXQUEUE, ffffd1022f512700.

```
0: kd> !ndiskd.cxadapter ffffd1022f1a0720 -basic -datapath


NETADAPTER

    Miniport           ffffd1022d048030 - Realtek PCIe GBE Family Controller NetAdapter Sample Driver #2
    NETADAPTER         00002efdd0e5f988
    WDFDEVICE          00002efdcf45f2f8

    Event Callbacks                        Function pointer   Symbol (if available)
    EvtAdapterCreateTxQueue                fffff80034151508   RtEthSample+1508
    EvtAdapterCreateRxQueue                fffff800341510ec   RtEthSample+10ec


DATAPATH QUEUES

    NETTXQUEUE         ffffd1022f512700
    NETRXQUEUE         ffffd1022cc7b0d0
```

By clicking on the NETTXQUEUE's handle or entering the **!ndiskd.netqueue -handle** command on the command line, you can see details for this queue, including the handle to its companion WDF object, the handle to its ring buffer, and function pointers for its registered callbacks.

```
0: kd> !ndiskd.netqueue ffffd1022f512700

    NETTXQUEUE         00002efdd0aed9a8
    Ring buffer        ffffd1022d000000

    Switch to EC thread

    Event Callbacks                        Function pointer   Symbol (if available)
    EvtQueueAdvance                        fffff80034152af8   RtEthSample+2af8
    EvtQueueArmNotification                fffff80034159a94   RtEthSample+9a94
    EvtQueueCancel                         fffff800341598d8   RtEthSample+98d8
```

See also

Network Driver Design Guide

Windows Vista and Later Networking Reference

Debugging the Network Stack

NDIS extensions (Ndiskd.dll)

!ndiskd.help

Network Adapter WDF Class Extension (Cx)

Summary of Objects

!ndiskd.netadapter

!ndiskd.cxadapter

!ndiskd.netrb

10/23/2019 • 2 minutes to read • Edit Online

The **!ndiskd.netrb** extension displays information about a NET_RING_BUFFER structure.

For more information about the Network Adapter WDF Class Extension (NetAdapterCx), see Network Adapter WDF Class Extension (Cx).

```
!ndiskd.netrb [-handle <x>] [-basic] [-dump] [-elementtype <str>]
```

Parameters

-handle
Required. Address of a NET_RING_BUFFER.

-basic
Displays basic information.

-dump
Displays information about each element in the NET_RING_BUFFER.

-elementtype
A string for the data type to use when referring to a ring buffer element.

DLL

Ndiskd.dll

Examples

Note See Summary of Objects to see a diagram explaining the relationship of the NET_RING_BUFFER object with other objects in the NetAdapterCx.

To obtain a handle for a NET_RING_BUFFER, follow these steps:

1. Run the **!ndiskd.netadapter** extension.
2. Click on the handle for a NetAdapter that has a NetAdapterCx driver installed.
3. Click the "More Information" link to the right of the NetAdapter's NETADAPTER object to run the **!ndiskd.cxadapter** extension.
4. Enter the **!ndiskd.cxadapter** command with the *-datapath* parameter to see that NETADAPTER's datapath queues.
5. Click on the handle for one of the datapath queues.

For details on Steps 1-4 of this procedure, see the examples on the **!ndiskd.cxadapter** topic. For details on Step 5 of this procedure, see the examples on the **!ndiskd.netqueue** topic. In the following example, look for the handle for this NETTXQUEUE's ring buffer, ffffd1022d000000.

```
0: kd> !ndiskd.netqueue ffffd1022f512700

    NETTXQUEUE          00002efdd0aed9a8
    Ring buffer         ffffd1022d000000

    Switch to EC thread

    Event Callbacks                         Function pointer    Symbol (if available)
    EvtQueueAdvance                         fffff80034152af8    RtEthSample+2af8
    EvtQueueArmNotification                 fffff80034159a94    RtEthSample+9a94
    EvtQueueCancel                          fffff800341598d8    RtEthSample+98d8
```

By clicking on the handle for the ring buffer or by entering the **!ndiskd.netrb -handle** command on the command line, you can see details for this NET_RING_BUFFER, including how many elements it contains and the address of its Begin and End indices.

```
0: kd> !ndiskd.netrb ffffd1022d000000

    NET_RING    ffffd1022d000000

    Number of elements 0x080
    Owned by OS         0x080
    Owned by Client     00000

    Begin Index         0x078 (ffffd1022d003c40 - NET_PACKET)
    Next Index          0x078 (ffffd1022d003c40 - NET_PACKET)
    End Index           0x078 (ffffd1022d003c40 - NET_PACKET)

    List all elements
```

To see this NET_RING_BUFFER's elements, either click the "List all elements" link at the bottom of its details or enter the **!ndiskd.netrb -dump** command on the command line. The following example has had the middle elements excised for brevity.

```
0: kd> !ndiskd.netrb ffffd1022d000000 -dump

    [000] ffffd1022d000040 - NET_PACKET
    [001] ffffd1022d0000c0 - NET_PACKET
    [002] ffffd1022d000140 - NET_PACKET
    [003] ffffd1022d0001c0 - NET_PACKET
    [004] ffffd1022d000240 - NET_PACKET
    [005] ffffd1022d0002c0 - NET_PACKET

    ...

    [07b] ffffd1022d003dc0 - NET_PACKET
    [07c] ffffd1022d003e40 - NET_PACKET
    [07d] ffffd1022d003ec0 - NET_PACKET
    [07e] ffffd1022d003f40 - NET_PACKET
    [07f] ffffd1022d003fc0 - NET_PACKET
```

See also

Network Driver Design Guide

Windows Vista and Later Networking Reference

Debugging the Network Stack

NDIS extensions (Ndiskd.dll)

!ndiskd.help

Network Adapter WDF Class Extension (Cx)

Summary of Objects

NET_RING_BUFFER

!ndiskd.netadapter

!ndiskd.cxadapter

!ndiskd.netqueue

!ndiskd.netpacket

10/23/2019 • 2 minutes to read • Edit Online

The **!ndiskd.netpacket** extension displays information about a NET_PACKET structure.

For more information about the Network Adapter WDF Class Extension (NetAdapterCx), see Network Adapter WDF Class Extension (Cx).

```
!ndiskd.netpacket [-handle <x>] [-basic] [-layout] [-checksum] [-data]
```

Parameters

-handle
Required. Address of a NET_PACKET.

-basic
Displays basic information.

-layout
Displays packet protocol layout.

-checksum
Displays packet checksum information.

-data
Dumps the payload memory.

DLL

Ndiskd.dll

Examples

Note See Summary of Objects to see a diagram explaining the relationship of the NET_PACKET object with other objects in the NetAdapterCx.

To obtain a handle for a NET_PACKET, follow these steps:

1. Run the **!ndiskd.netadapter** extension.
2. Click on the handle for a NetAdapter that has a NetAdapterCx driver installed.
3. Click the "More Information" link to the right of the NetAdapter's NETADAPTER object to run the **!ndiskd.cxadapter** extension.
4. Enter the **!ndiskd.cxadapter** command with the *-datapath* parameter to see that NETADAPTER's datapath queues.
5. Click on the handle for one of the datapath queues.
6. Click on the handle for that datapath queue's ring buffer.
7. Click on the "List all elements" link at the bottom of the ring buffer details to see the elements it contains.

For details on Steps 1-4 of this procedure, see the examples on the **!ndiskd.cxadapter** topic. For details on Step 5 of this procedure, see the examples on the **!ndiskd.netqueue** topic. For details on Steps 6-7 of this procedure, see the examples on the **!ndiskd.netrb** topic. In the following example, look for the handle for the first NET_PACKET, ffffd1022d000040.

```
0: kd> !ndiskd.netrb ffffd1022d000000 -dump

    [000] ffffd1022d000040 - NET_PACKET
    [001] ffffd1022d0000c0 - NET_PACKET
    [002] ffffd1022d000140 - NET_PACKET
    [003] ffffd1022d0001c0 - NET_PACKET
    [004] ffffd1022d000240 - NET_PACKET
    [005] ffffd1022d0002c0 - NET_PACKET

    ...

    [07b] ffffd1022d003dc0 - NET_PACKET
    [07c] ffffd1022d003e40 - NET_PACKET
    [07d] ffffd1022d003ec0 - NET_PACKET
    [07e] ffffd1022d003f40 - NET_PACKET
    [07f] ffffd1022d003fc0 - NET_PACKET
```

By clicking on the handle for this NET_PACKET or by entering **!ndiskd.netpacket -handle** on the command line, you can see details for this NET_PACKET, including the ring buffer that contains it, the datapath queue that contains its ring buffer, and the handle for its first fragment.

```
0: kd> !ndiskd.netpacket ffffd1022d000040


    NET_PACKET         ffffd1022d000040    Ring Buffer        ffffd1022d000000
    First fragment     ffffd1022d000040    NETTXQUEUE         ffffd1022f512700

    Client Context     ffffd1022d000090

    Show protocol layout
    Show checksum information
    Dump data payload
```

You can now combine the basic description with any of the other **!ndiskd.netpacket** parameters, or all of them, to see specific information for this fragment. The following example uses all parameters.

```
0: kd> !ndiskd.netpacket ffffd1022d000040 -basic -layout -checksum -data

    NET_PACKET          ffffd1022d000040    Ring Buffer         ffffd1022d000000
    First fragment      ffffd1022d000040    NETTXQUEUE          ffffd1022f512700

    Client Context      ffffd1022d000090


    Protocol Layout

    Layer 2 Type        ETHERNET
    Header Length       0n14

    Layer 3 Type        IPV4_NO_OPTIONS
    Header Length       0n20

    Layer 4 Type        UDP
    Header Length       8


    Checksum Information

    Layer 2             TX_PASSTHROUGH
    Layer 3             TX_REQUIRED
    Layer 4             TX_PASSTHROUGH


    Payload data

    Fragment            ffffd1022d000040
    ffffd102303e8332  00 00 01 02 71 68 0a 89-be 39 e0 00 00 16 94 04  ····qh···9·····
    ffffd102303e8342  00 00 22 00 fa 01 00 00-00 01 03 00 00 00 e0 00  ··"·············
    ffffd102303e8352  00 fc
```

See also

Network Driver Design Guide

Windows Vista and Later Networking Reference

Debugging the Network Stack

NDIS extensions (Ndiskd.dll)

!ndiskd.help

Network Adapter WDF Class Extension (Cx)

Summary of Objects

NET_PACKET

!ndiskd.netadapter

!ndiskd.cxadapter

!ndiskd.netqueue

!ndiskd.netrb

!ndiskd.netpacketfragment

10/23/2019 • 2 minutes to read • Edit Online

The **!ndiskd.netpacketfragment** extension displays information about a NET_PACKET_FRAGMENT structure.

For more information about the Network Adapter WDF Class Extension (NetAdapterCx), see Network Adapter WDF Class Extension (Cx).

```
!ndiskd.netpacketfragment [-handle <x>]
```

Parameters

-handle
Required. Address of a NET_PACKET_FRAGMENT.

DLL

Ndiskd.dll

Examples

Note See Summary of Objects to see a diagram explaining the relationship of the NET_PACKET object with other objects in the NetAdapterCx.

To obtain a handle for a NET_PACKET, follow these steps:

1. Run the **!ndiskd.netadapter** extension.
2. Click on the handle for a NetAdapter that has a NetAdapterCx driver installed.
3. Click the "More Information" link to the right of the NetAdapter's NETADAPTER object to run the **!ndiskd.cxadapter** extension.
4. Enter the **!ndiskd.cxadapter** command with the *-datapath* parameter to see that NETADAPTER's datapath queues.
5. Click on the handle for one of the datapath queues.
6. Click on the handle for that datapath queue's ring buffer.
7. Click on the "List all elements" link at the bottom of the ring buffer details to see the elements it contains.
8. Click on one of the NET_PACKET objects in the ring buffer's list of elements.

For details on Steps 1-4 of this procedure, see the examples on the **!ndiskd.cxadapter** topic. For details on Step 5 of this procedure, see the examples on the **!ndiskd.netqueue** topic. For details on Steps 6-7 of this procedure, see the examples on the **!ndiskd.netrb** topic. For details on Step 8 of this procedure, see the examples on the **!ndiskd.netpacket** topic. In the following example, look for the handle for the first fragment of this NET_PACKET, ffffd1022d000040.

```
0: kd> !ndiskd.netpacket ffffd1022d000040


    NET_PACKET         ffffd1022d000040    Ring Buffer        ffffd1022d000000
    First fragment     ffffd1022d000040    NETTXQUEUE         ffffd1022f512700

    Client Context     ffffd1022d000090

    Show protocol layout
    Show checksum information
    Dump data payload
```

By clicking on the handle for the first fragment or by entering the **!ndiskd.netpacketfragment -handle** command on the command line, you can see details for this NET_PACKET_FRAGMENT, including its Virtual Address, capacity, and whether or not it is the last packet in the NET_PACKET chain of fragments.

```
0: kd> !ndiskd.netpacketfragment ffffd1022d000040

    NET_PACKET_FRAGMENT ffffd1022d000040

    Virtual Address    ffffd102303e82f8
    Capacity           0n92
    Valid Length       0n34
    Offset             0n58

    Last packet of chain
```

See also

Network Driver Design Guide

Windows Vista and Later Networking Reference

Debugging the Network Stack

NDIS extensions (Ndiskd.dll)

!ndiskd.help

Network Adapter WDF Class Extension (Cx)

Summary of Objects

NET_PACKET_FRAGMENT

NET_PACKET

!ndiskd.netadapter

!ndiskd.cxadapter

!ndiskd.netqueue

!ndiskd.netrb

!ndiskd.netpacket

!ndiskd.oid

10/23/2019 • 3 minutes to read • Edit Online

The **!ndiskd.oid** extension displays information about an NDIS OID request. If you run this extension with no parameters, !ndiskd will display a list of all pending OID requests on all miniports and filters. Each miniport or filter has at most one pending OID request and any number of queued OID requests.

Note that filters typically clone OID requests and pass the clone down. This means that even if a protocol issues a single OID request, there may be multiple instances of cloned requests: one in each filter and another in the miniport. **!ndiskd.oid** will show each clone separately, so you may see more pending OIDs than the protocol has actually issued.

```
!ndiskd.oid [-handle <x>] [-legacyoid] [-nolimit>] [-miniport <x>]
```

Parameters

-handle
Handle of an NDIS_OID_REQUEST

-legacyoid
Treats as a legacy NDIS_REQUEST instead of an NDIS_OID_REQUEST.

-nolimit
Does not limit the number of pending OIDs that are displayed.

-miniport
Finds pending OID requests on this miniport's stack.

DLL

Ndiskd.dll

Remarks

!ndiskd.oid shows you a list of all the pending OIDs on the system at a time, so it can be helpful in debugging system hangs or 0x9F bug check situations (DRIVER_POWER_STATE_FAILURE). For example, suppose analyzing a fictitious 0x9F bug check revealed that the system was hung on an IRP and was waiting for NDIS. In NDIS, IRPs from the OS are translated into OIDs, including power transitions, so by running **!ndiskd.oid** you could see that, in this example, a device at the bottom of the stack might have been clinging to an OID_PNP_SET_POWER and hung the rest of the stack. NDIS drivers should not pend an OID for more than one second, so you could then investigate why that device kept the OID pending for too long to try to solve the issue.

Examples

To see an example of pending OIDS on a system that is running normally, set a breakpoint on a miniport's OID request handler routine (in the miniport's corresponding miniport driver). First, run the **!ndiskd.minidriver** command with no parameters to get a list of miniport drivers on the system. In this example output, look for the handle for the kdnic minidriver, ffffdf801418d650..

```
3: kd> !ndiskd.minidriver
    ffffdf8015a98380 - tunnel
    ffffdf801418d650 - kdnic
```

Click on the handle for the minidriver, then click on the "Handlers" link at the bottom of its details page to see the list of its handlers. You can alternatively enter the **!ndiskd.minidriver -handle -handlers** command. Once you have the list of the minidriver's handlers, look for the OidRequestHandler, whose handle is fffff80f1fd71c90 in this example.

```
2: kd> !ndiskd.minidriver ffffdf801418d650 -handlers


HANDLERS

    NDIS Handler                        Function pointer   Symbol (if available)
    InitializeHandlerEx                 fffff80f1fd78230  bp
    SetOptionsHandler                   fffff80f1fd72800  bp
    HaltHandlerEx                       fffff80f1fd78040  bp
    ShutdownHandlerEx                   fffff80f1fd722c0  bp

    CheckForHangHandlerEx               fffff80f1fd72810  bp
    ResetHandlerEx                      fffff80f1fd72f70  bp

    PauseHandler                        fffff80f1fd78000  bp
    RestartHandler                      fffff80f1fd78940  bp

    OidRequestHandler                   fffff80f1fd71c90  bp
    CancelOidRequestHandler             fffff80f1fd722c0  bp
    DirectOidRequestHandler             [None]
    CancelDirectOidRequestHandler       [None]
    DevicePnPEventNotifyHandler         fffff80f1fd789a0  bp

    SendNetBufferListsHandler           fffff80f1fd71870  bp
    ReturnNetBufferListsHandler         fffff80f1fd71b50  bp
    CancelSendHandler                   fffff80f1fd722c0  bp
```

Now either click on the "bp" link to the right of the OidRequestHandler or enter the **bp -handle** command with its handle to set a breakpoint on that routine. Next, type the **g** command to allow your debugee target machine to run and hit the breakpoint you just set.

```
2: kd> bp fffff80f1fd71c90
2: kd> g
Breakpoint 1 hit
fffff80f`1fd71c90 448b4204        mov     r8d,dword ptr [rdx+4]
```

Once you have triggered the breakpoint on a minidriver's OID request handler routine as shown by the previous example, you can run the !ndiskd.oid command to see a list of all the pending OIDs on the system.

```
1: kd> !ndiskd.oid


ALL PENDING OIDs

    NetAdapter         ffffdf80140c71a0 - Microsoft Kernel Debug Network Adapter
        Current OID        OID_GEN_STATISTICS
    Filter             ffffdf8014950c70 - Microsoft Kernel Debug Network Adapter-WFP Native MAC Layer
LightWeight Filter-0000
        Current OID        OID_GEN_STATISTICS
    Filter             ffffdf801494dc70 - Microsoft Kernel Debug Network Adapter-QoS Packet Scheduler-0000
        Current OID        OID_GEN_STATISTICS
```

In this example, the OID pending is OID_GEN_STATISTICS. When you look at the results of !ndiskd.oid, recall that filters clone OID requests and pass them down the stack, and OIDs typically get passed from filter to filter to miniport. Therefore, although it may look like there are three separate OID requests with the same name in this example, there is actually one logical operation taking place which was physically spread across 3 OIDs and on 3 drivers.

See also

Network Driver Design Guide

Windows Vista and Later Networking Reference

Debugging the Network Stack

NDIS extensions (Ndiskd.dll)

!ndiskd.help

0x9F bug check

OID_PNP_SET_POWER

bp, bu, bm (Set Breakpoint)

OID_GEN_STATISTICS

NDIS OIDs

NDIS OID Request Interface

!ndiskd.interfaces

10/23/2019 • 2 minutes to read • Edit Online

The **!ndiskd.interfaces** extension displays information about a network interface. If you run this extension with no parameters, !ndiskd will display a list of all network interfaces.

For more information about network interfaces, see NDIS Network Interfaces.

```
!ndiskd.interfaces [-handle <x>] [-luid <x>]
```

Parameters

-handle
Handle of a network interface.

-luid
NetLuid (Net Locally Unique Identifier) of a network interface.

DLL

Ndiskd.dll

Examples

Run the **!ndiskd.interfaces** extension with no parameters to see a list of all network interfaces on the system. In this example, look for the Intel(R) 82579LM Gigabit Network Connection interface. Its handle is ffffdf80139f8a20.

```
1: kd> !ndiskd.interfaces
    Interface
    ffffdf801494fa20 - Microsoft Kernel Debug Network Adapter-WFP Native MAC Layer LightWeight Filter-0000
    ffffdf801494c010 - Microsoft Kernel Debug Network Adapter-QoS Packet Scheduler-0000
    ffffdf801494ba20 - Microsoft Kernel Debug Network Adapter-WFP 802.3 MAC Layer LightWeight Filter-0000
    ffffdf80139b3a20 - Microsoft Kernel Debug Network Adapter
    ffffdf80139f8a20 - Intel(R) 82579LM Gigabit Network Connection
    ffffdf80139baa20 - WAN Miniport (IP)
    ffffdf80139a4a20 - WAN Miniport (IPv6)
    ffffdf80131d0010 - WAN Miniport (Network Monitor)
    ffffdf80131cda20 - WAN Miniport (PPPOE)
    ffffdf80139b6a20 - Software Loopback Interface 1
    ffffdf80139b0a20 - Microsoft ISATAP Adapter
    ffffdf80139ada20 - WAN Miniport (SSTP)
    ffffdf80131cf010 - WAN Miniport (IKEv2)
    ffffdf80139fea20 - WAN Miniport (L2TP)
    ffffdf80139a7a20 - WAN Miniport (PPTP)
    ffffdf80139aaa20 - Microsoft ISATAP Adapter #2
    ffffdf80139fba20 - Teredo Tunneling Pseudo-Interface
```

By clicking on the handle for an interface or by entering the **!ndiskd.interfaces -handle** command, you can see the details about that interface, including its Identifier information and its current state. In this example, you can see that the Intel(R) 82579LM Gigabit Network Connection is an Ethernet connection (its ifAlias) and is in a MediaConnectUnknown state for its connection (as it has been reserved for use by the Windows kernel debugger).

```
1: kd> !ndiskd.interfaces ffffdf80139f8a20


INTERFACE

    Ethernet

    Ndis handle        ffffdf80139f8a20
    IfProvider         ffffdf80131ca8d0 - The NDIS interface provider
    Provider context   ffffdf80139f8a20

    ifType             IF_TYPE_ETHERNET_CSMACD
    Media type         802.3
    Physical medium    802.3
    Access type        BROADCAST
    Direction type     SEND_AND_RECEIVE
    Connection type    DEDICATED

    ifConnectorPresent No

    Network            ffffdf80139b8900 - [Unnamed network]
    Compartment        ffffdf80139b9940 - Compartment #1


IDENTIFIERS

    ifAlias            Ethernet
    ifDescr            Intel(R) 82579LM Gigabit Network Connection
    ifName (NET_LUID)  06:8001
    ifPhysAddress      18-03-73-c1-e8-72

    ifIndex            0n14
    ifGuid             cbddfde1-5570-4c65-9d47-52d63abce00c


STATE

    Connected          MediaConnectUnknown
    ifOperStatus       NOT_PRESENT

    Link speed         0 [Speed is not applicable]
    ifMtu              0
    Duplex             UnknownDuplex

    Refer to RFC 2863 for definitions of many of these terms
```

See also

Network Driver Design Guide

Windows Vista and Later Networking Reference

Debugging the Network Stack

NDIS extensions (Ndiskd.dll)

!ndiskd.help

NDIS Network Interfaces

NET_LUID Value

!ndiskd.ifprovider

10/23/2019 • 2 minutes to read • Edit Online

The **!ndiskd.ifprovider** extension displays information about an NDIS interface provider (IfProvider). If you run this extension with no parameters, !ndiskd will display a list of all registered NDIS interface providers.

```
!ndiskd.ifprovider [-handle <x>]
```

Parameters

-handle
Handle of an IfProvider.

DLL

Ndiskd.dll

Examples

Run the **!ndiskd.ifprovider** extension with no parameters to get a list of all registered IfProviders.

```
1: kd> !ndiskd.ifprovider
    IfProvider
    ffffd20d14334180 - wanarp
    ffffd20d1264a950 - wfplwfs
    ffffd20d11deae00 - The NDIS loopback provider
    ffffd20d11deae70 - The NDIS interface provider
```

You can see from the previous example that the debugee machine has four interface providers registered. Two of them are NDIS interface providers.

Note Interface providers are a generic concept and aren't required to be miniport drivers. While a miniport driver may choose to register as an interface provider if desired, most miniport drivers do not do so because NDIS has a built-in interface provider. The NDIS built-in interface provider automatically provides interfaces for every miniport driver, every Light-Weight Filter (LWF) module, and the loopback interface. For more information, see NDIS interface provider.

The following example shows the details for the "wanarp" interface provider in the previous example, whose handle is ffffd20d14334180.

```
1: kd> !ndiskd.ifprovider ffffd20d14334180


IF PROVIDER

    wanarp
    Ndis handle        ffffd20d14334180


INTERFACES

    Interface
    [No interfaces found]


HANDLERS

    Protocol handler                    Function pointer   Symbol (if available)
    QueryObjectHandler                  fffff80d2f0414b0  bp wanarp!WanNdisIfQueryHandler
    SetObjectHandler                    fffff80d2f04bd10  bp wanarp!WanNdisIfSetHandler
```

See also

Network Driver Design Guide

Windows Vista and Later Networking Reference

Debugging the Network Stack

NDIS extensions (Ndiskd.dll)

!ndiskd.help

Registering as an Interface Provider

!ndiskd.ifstacktable

10/23/2019 • 2 minutes to read • Edit Online

The **!ndiskd.ifstacktable** extension displays the network interface stack table (ifStackTable).

For more information about the interface stack table, see Maintaining a Network Interface Stack.

```
!ndiskd.ifstacktable
```

Parameters

This extension has no parameters.

DLL

Ndiskd.dll

Examples

Run the **!ndiskd.ifstacktable** command to see the ifStackTable.

```
3: kd> !ndiskd.ifstacktable


INTERFACE STACK TABLE

    Lower interface    Lower IfIndex       Higher IfIndex     Higher interface
    ffffdf80139b3a20   6                   15                 ffffdf801494fa20
    ffffdf801494fa20   15                  16                 ffffdf801494c010
    ffffdf801494c010   16                  17                 ffffdf801494ba20
```

NDIS maintains the stack table for NDIS miniport adapters, NDIS 5.x filter intermediate drivers, and NDIS filter modules, whereas NDIS MUX Intermediate Drivers drivers are required to specify the internal interface relationship between the virtual miniport interface and the protocol lower interface. Therefore, the ifStackTable could be useful for seeing the interface stack relationships in a system with more complicated MUX drivers installed.

Since there are no NDIS MUX Intermediate drivers installed on this example system, the ifStackTable only shows the stack relationships that NDIS has provided. In the following example, clicking on the handle for the Lower interface of the third row (handle ffffdf801494c010, Lower IfIndex 16) shows the interface for the QoS Packet Scheduler.

```
3: kd> !ndiskd.interface ffffdf801494c010


INTERFACE

    [Zero-length string]

    Ndis handle        ffffdf801494c010
    IfProvider         ffffdf80131ca8d0 - The NDIS interface provider
    NDIS filter        ffffdf801494dc70 - Microsoft Kernel Debug Network Adapter-QoS Packet Scheduler-0000

    ifType             IF_TYPE_ETHERNET_CSMACD
    Media type         802.3
    Physical medium    NdisPhysicalMediumOther
    Access type        BROADCAST
    Direction type     SEND_AND_RECEIVE
    Connection type    DEDICATED

    ifConnectorPresent No

    Network            ffffdf80139b8900 - [Unnamed network]
    Compartment        ffffdf80139b9940 - Compartment #1


IDENTIFIERS

    ifAlias            [Zero-length string]
    ifDescr            Microsoft Kernel Debug Network Adapter-QoS Packet Scheduler-0000
    ifName (NET_LUID)  06:01
    ifPhysAddress      18-03-73-c1-e8-72

    ifIndex            0n16
    ifGuid             fc2a0ae1-b103-11e6-b724-806e6f6e6963


STATE

    Connected          Connected
    ifOperStatus       DORMANT
    ifOperStatusFlags  DORMANT_PAUSED

    Link speed         1000000000 (1 Gbps)
    ifMtu              0n1500
    Duplex             FullDuplex

    Refer to RFC 2863 for definitions of many of these terms
```

Continuing the same example, clicking the handle for the Higher interface of the third row (handle ffffdf801494ba20, Higher IfIndex 17) shows the interface for the WFP 802.3 MAC Layer LightWeight Filter.

```
3: kd> !ndiskd.interface ffffdf801494ba20


INTERFACE

    [Zero-length string]

    Ndis handle        ffffdf801494ba20    [type it]
    IfProvider         ffffdf80131ca8d0 - The NDIS interface provider
    NDIS filter        ffffdf801494c670 - Microsoft Kernel Debug Network Adapter-WFP 802.3 MAC Layer
LightWeight Filter-0000

    ifType             IF_TYPE_ETHERNET_CSMACD
    Media type         802.3
    Physical medium    NdisPhysicalMediumOther
    Access type        BROADCAST
    Direction type     SEND_AND_RECEIVE
    Connection type    DEDICATED

    ifConnectorPresent No

    Network            ffffdf80139b8900 - [Unnamed network]
    Compartment        ffffdf80139b9940 - Compartment #1


IDENTIFIERS

    ifAlias            [Zero-length string]
    ifDescr            Microsoft Kernel Debug Network Adapter-WFP 802.3 MAC Layer LightWeight Filter-0000
    ifName (NET_LUID)  06:02
    ifPhysAddress      18-03-73-c1-e8-72

    ifIndex            0n17
    ifGuid             fc2a0ae0-b103-11e6-b724-806e6f6e6963


STATE

    Connected          Connected
    ifOperStatus       DORMANT
    ifOperStatusFlags  DORMANT_PAUSED

    Link speed         1000000000 (1 Gbps)
    ifMtu              0n1500
    Duplex             FullDuplex

    Refer to RFC 2863 for definitions of many of these terms
```

This shows that the WFP 802.3 MAC Layer LightWeight Filter sits above the QoS Packet Scheudler filter in the network interface stack. You can confirm this by running the **!ndiskd.netreport** extension, which shows you the network stack visually.

See also

Network Driver Design Guide

Windows Vista and Later Networking Reference

Debugging the Network Stack

NDIS extensions (Ndiskd.dll)

!ndiskd.help

Maintaining a Network Interface Stack

NDIS MUX Intermediate Drivers

!ndiskd.netreport

!ndiskd.compartments

10/23/2019 • 2 minutes to read • Edit Online

The **!ndiskd.compartments** extension displays all network compartments.

```
!ndiskd.compartments
```

Parameters

This extension has no parameters.

DLL

Ndiskd.dll

Remarks

Compartments are a way that NDIS manages interfaces. Third party interface providers only use the primary compartment, as described in the **CompartmentId** member of the **NDIS_BIND_PARAMETERS** structure.

Examples

Run the **!ndiskd.compartments** extension to see a list of all network compartments. In this example, there is only one compartment (the primary one).

```
3: kd> !ndiskd.compartments
    Compartment        ffffdf80139b9940
    ID                 1
    Loopback Network   ffffdf80139b8900
    Loopback Interface ffffdf80139b6a20
    Networks:
                       ffffdf80139b8900    [Unnamed network]
```

See also

Network Driver Design Guide

Windows Vista and Later Networking Reference

Debugging the Network Stack

NDIS extensions (Ndiskd.dll)

!ndiskd.help

NDIS_BIND_PARAMETERS

!ndiskd.pkt

6/25/2019 • 2 minutes to read • Edit Online

Warning This extension is for legacy NDIS 5.x drivers. The NDIS_PACKET structure and its associated architecture have been deprecated.

The **!ndiskd.pkt** extension displays information about an NDIS_PACKET structure.

```
!ndiskd.pkt [-packet] [-verbosity]
```

Parameters

Packet
Specifies the address of the packet.

Verbosity
Specifies the amount of detail to be displayed.

DLL

Ndiskd.dll

See also

NDIS_PACKET

!ndiskd.pktpools

6/25/2019 • 2 minutes to read • Edit Online

Warning This extension is for legacy NDIS 5.x drivers. The NDIS_PACKET structure and its associated architecture have been deprecated.

The **!ndiskd.pktpools** extension displays a list of all allocated packet pools.

```
!ndiskd.pktpools
```

DLL

Ndiskd.dll

Examples

Run the **!ndiskd.pktpools** extension to see a list of all allocated packet pools on the system. Note that the handles for the packet pools are not clickable, which means you can't explore further information about the packet pool. This is because NDIS does not use packet pools starting with NDIS 6.0, so these pools are allocated only for legacy drivers which may still be on older systems. The debugee machine in this example does not have any legacy NDIS 5.x drivers installed so the packet pools are not used. This example is for illustrative purposes only.

```
3: kd> !ndiskd.pktpools
Pool      Allocator  BlocksAllocated  BlockSize  PktsPerBlock  PacketLength
ffffdf80131d58c0  fffff80f1fbe3e8f   0x1          0x1000     0xa           0x190   ndis!DriverEntry+6af
ffffdf80131d5940  fffff80f1fbe3e71   0x1          0x1000     0xa           0x180   ndis!DriverEntry+691
```

See also

Windows 2000 and Windows XP Networking Design Guide

Windows 2000 and Windows XP Networking Reference

NDIS_PACKET

!ndiskd.findpacket

6/25/2019 • 2 minutes to read • Edit Online

Warning This extension is for legacy NDIS 5.x drivers. The NDIS_PACKET structure and its associated architecture have been deprecated.

The **!ndiskd.findpacket** extension finds the specified packets.

```
!ndiskd.findpacket [-VirtualAddress] [-PoolAddress]
```

Parameters

VirtualAddress
Specifies a virtual address that is contained in the desired packet.

PoolAddress
Specifies a pool address. All unreturned packets in this pool will be displayed.

DLL

Ndiskd.dll

See also

NDIS_PACKET

!ndiskd.vc

10/23/2019 • 2 minutes to read • Edit Online

The **!ndiskd.vc** extension displays a Connection-Oriented (CoNDIS) virtual connection (VC).

```
!ndiskd.vc [-handle <x>]
```

Parameters

-handle
Required. Handle of a VC pointer.

DLL

Ndiskd.dll

Remarks

For more information about CoNDIS, see Connection-Oriented NDIS.

For more information about CoNDIS virtual connections, see Virtual Connections.

Examples

CoNDIS is used in certain situations such as connecting to a VPN, so running **!ndiskd.vc** will not show you results unless a miniport driver on your system has created and activated a CoNDIS virtual connection. The following example shows results from a machine that is connected to a VPN network. First, run the **!ndiskd.netadapter** extension with no parameters to see a list of miniports and miniport drivers on the system. In the following output, look for the miniport driver for the Marvell AVASTAR Wireless-AC Network Controller network adapter. Its handle is ffffc804af2e3710.

```
1: kd> !ndiskd.netadapter
    Driver             NetAdapter          Name
    ffffc804af2e3710   ffffc804b9e6f1a0    Marvell AVASTAR Wireless-AC Network Controller
    ffffc804b99b9020   ffffc804b9c301a0    WAN Miniport (Network Monitor)
    ffffc804b99b9020   ffffc804b9c2a1a0    WAN Miniport (IPv6)
    ffffc804b99b9020   ffffc804b8a8a1a0    WAN Miniport (IP)
    ffffc804ae9d7020   ffffc804b9ceb1a0    WAN Miniport (PPPOE)
    ffffc804b9ca5900   ffffc804b9e601a0    WAN Miniport (PPTP)
    ffffc804b99dc720   ffffc804b99b01a0    WAN Miniport (L2TP)
    ffffc804b86581b0   ffffc804b9c6c1a0    WAN Miniport (IKEv2)
    ffffc804ad4a7250   ffffc804b99651a0    WAN Miniport (SSTP)
    ffffc804b11c4020   ffffc804b85821a0    Microsoft ISATAP Adapter
    ffffc804b11c4020   ffffc804b71731a0    Microsoft ISATAP Adapter #2
    ffffc804ad725020   ffffc804b05e71a0    Surface Ethernet Adapter #2
    ffffc804b0bf0020   ffffc804b0c011a0    Bluetooth Device (Personal Area Network)
    ffffc804aef695e0   ffffc804aed331a0    TAP-Windows Adapter V9
```

Next, enter the **!ndiskd.vc** command with the miniport driver's handle to see the virtual connections opened by that miniport driver.

```
1: kd> !ndiskd.vc ffffc804af2e3710


VIRTUAL CALL

    [Zero-length string]
    Ndis handle        ffffc804af2e3710
    VC Index           0

    AF                 fffff80965fd5888
    Call Flags         [No flags set]
    VC Flags           [Unrecognized flags 04a80100] VC_ACTIVATE_PENDING

    Miniport           fffff80965ffaa20 - [Invalid NetAdapter]
    Miniport Context   fffff80965ffaad0

    Call Manager       fffff80965ff9b50 - [Invalid Open]
    Call Manager Context fffff80965ff9c70

    Client             ffffc804af96fd78 - [Invalid Open]
    Client Context     00003206
```

See also

Network Driver Design Guide

Windows Vista and Later Networking Reference

Debugging the Network Stack

NDIS extensions (Ndiskd.dll)

!ndiskd.help

Connection-Oriented NDIS

Virtual Connections

!ndiskd.netadapter

!ndiskd.af

10/23/2019 • 2 minutes to read • Edit Online

The **!ndiskd.af** extension displays a Connection-Oriented NDIS (CoNDIS) address family (AF).

```
!ndiskd.af [-handle <x>]
```

Parameters

-handle
Required. Handle of a CoNDIS address family.

DLL

Ndiskd.dll

Remarks

For more information about CoNDIS, see Connection-Oriented NDIS.

For more information about CoNDIS address families, see Address Families.

Examples

CoNDIS is used in certain situations such as connecting to a VPN, so running **!ndiskd.af** will not show you results unless a miniport driver on your system has created and activated a CoNDIS virtual connection. The following example shows results from a machine that is connected to a VPN network. First, run the **!ndiskd.netadapter** extension with no parameters to see a list of miniports and miniport drivers on the system. In the following output, look for the miniport driver for the Marvell AVASTAR Wireless-AC Network Controller network adapter. Its handle is ffffc804af2e3710.

```
1: kd> !ndiskd.netadapter
    Driver             NetAdapter          Name
    ffffc804af2e3710   fffc804b9e6f1a0    Marvell AVASTAR Wireless-AC Network Controller
    ffffc804b99b9020   fffc804b9c301a0    WAN Miniport (Network Monitor)
    ffffc804b99b9020   fffc804b9c2a1a0    WAN Miniport (IPv6)
    ffffc804b99b9020   fffc804b8a8a1a0    WAN Miniport (IP)
    ffffc804ae9d7020   fffc804b9ceb1a0    WAN Miniport (PPPOE)
    ffffc804b9ca5900   fffc804b9e601a0    WAN Miniport (PPTP)
    ffffc804b99dc720   fffc804b99b01a0    WAN Miniport (L2TP)
    ffffc804b86581b0   fffc804b9c6c1a0    WAN Miniport (IKEv2)
    ffffc804ad4a7250   fffc804b99651a0    WAN Miniport (SSTP)
    ffffc804b11c4020   fffc804b85821a0    Microsoft ISATAP Adapter
    ffffc804b11c4020   fffc804b71731a0    Microsoft ISATAP Adapter #2
    ffffc804ad725020   fffc804b05e71a0    Surface Ethernet Adapter #2
    ffffc804b0bf0020   fffc804b0c011a0    Bluetooth Device (Personal Area Network)
    ffffc804aef695e0   fffc804aed331a0    TAP-Windows Adapter V9
```

Next, enter the **!ndiskd.af** command with the miniport driver's handle to see the address family for this miniport driver, which is acting as a connection-oriented client.

```
1: kd> !ndiskd.af ffffc804af2e3710


ADDRESS FAMILY

    Ndis handle        ffffc804af2e3710
    Flags              [Unrecognized flags 399b9020] AF_CLOSING
    References         ffffc804
    Close Requested?   0

    Miniport           0 - [Unreadable NetAdapter]

    Call Manager       ffffc804b90a4ac0 - [Invalid Open]
    Call Manager Context 007a0078

    Client             00060000 - [Unreadable Open]
    Client Context     ffffc804af2e3888


CLIENT HANDLERS

    Client Handler                      Function pointer   Symbol (if available)
    ClCreateVcHandler                   fffff80965fd5888   mrvlpcie8897!Globals+8
    ClDeleteVcHandler                   [None]
    ClRequestHandler                    ffffc804af96fd78
    ClRequestCompleteHandler            ffffc804af96fd78
    ClOpenAfCompleteHandler             [None]
    ClCloseAfCompleteHandler            [None]
    ClRegisterSapCompleteHandler        000132060098028a
    ClDeregisterSapCompleteHandler      [None]
    ClMakeCallCompleteHandler           fffff80965ff9ec0   wdiwifi!MPWrapperSetOptions
    ClCloseCallCompleteHandler          fffff80965ff9c70   wdiwifi!MPWrapperHalt
    ClModifyCallQoSCompleteHandler      fffff80965ff9b50   wdiwifi!MPWrapperInitializeEx
    ClAddPartyCompleteHandler           fffff80965e71a08   mrvlpcie8897!MPUnload
    ClDropPartyCompleteHandler          fffff80965ffa070   wdiwifi!MPWrapperPause
    ClIncomingDropPartyHandler          fffff80965ffaa20   wdiwifi!MPWrapperReturnNetBufferLists
    ClIncomingCallHandler               fffff80965ffa1e0   wdiwifi!MPWrapperRestart
    ClCallConnectedHandler              fffff80965ffaad0   wdiwifi!MPWrapperCancelSendNetBufferLists
    ClIncomingCloseCallHandler          fffff80965ffa870   wdiwifi!MPWrapperSendNetBufferLists
    ClIncomingCallQoSChangeHandler      fffff80965ffa610   wdiwifi!MPWrapperOidRequest
```

See also

Network Driver Design Guide

Windows Vista and Later Networking Reference

Debugging the Network Stack

NDIS extensions (Ndiskd.dll)

!ndiskd.help

Connection-Oriented NDIS

Address Families

!ndiskd.ndisref

10/23/2019 • 2 minutes to read • Edit Online

The **!ndiskd.ndisref** extension displays a debug log of a tracked reference count.

```
!ndiskd.ndisref [-handle <x>] [-tagtype <str>] [-stacks] [-tag <str>] [-refdebug]
```

Parameters

-handle
Required. Handle of the refcount block.

-tagtype
Enum type of the tags.

-stacks
Includes stack traces (if available).

-tag
Limits display to a single tag.

-refdebug
Shows detailed debug log, if available.

DLL

Ndiskd.dll

Examples

The following example passes the handle of an NDIS miniport driver to the **!ndiskd.ndisref** extension to display the refcount block for that driver. First, run **!ndiskd.minidriver** with no parameters to see a list of all miniport drivers on the system. In the example output below, look for the handle for the kdnic driver, ffffdf801418d650.

```
3: kd> !ndiskd.minidriver
    ffffdf8015a98380 - tunnel
    ffffdf801418d650 - kdnic
```

Using the miniport driver's handle, enter the **!ndiskd.ndisref -handle** command to see the refcount block for this miniport driver. The following example has the middle of the refcount block excised for brevity.

```
3: kd> !ndiskd.ndisref ffffdf801418d650


REFCOUNT BLOCK

    Tag                                 Number of references
    0n0                                 0n6293              Show stacks
    0n1                                 0n1045              Show stacks
    0n2                                 0n4294966717        Show stacks
    0n5                                 0n25048             Show stacks
    0n18                                0n4294967293        Show stacks
    0n19                                0n4294967295        Show stacks
    0n21                                0n4294967036        Show stacks
    0n23                                0n30818             Show stacks
    0n24                                0n24693             Show stacks
    0n25                                0n25808             Show stacks

...

    0n153                               0n7                 Show stacks
    0n154                               0n3                 Show stacks
    0n156                               0n29972             Show stacks
    0n159                               0n4294959128        Show stacks
    0n160                               0n30892             Show stacks
    0n161                               0n136               Show stacks
    0n162                               0n4294951910        Show stacks
    0n163                               0n30892             Show stacks
    0n164                               0n136               Show stacks
    0n167                               0n4294965996        Show stacks

    Include inactive tags
```

See also

Network Driver Design Guide

Windows Vista and Later Networking Reference

Debugging the Network Stack

NDIS extensions (Ndiskd.dll)

!ndiskd.help

!ndiskd.minidriver

!ndiskd.ndisevent

10/23/2019 • 2 minutes to read • Edit Online

Note Third party network driver developers are not expected to manually use this extension command. You can run it to see the information it displays but you are not able to reuse the details it provides in your driver.

The **!ndiskd.ndisevent** extension displays an NDIS debug event log.

```
!ndiskd.ndisevent [-handle <x>] [-tagtype <str>]
```

Parameters

-handle
Required. Handle of the event log.

-tagtype
Enum type of the tags.

DLL

Ndiskd.dll

Examples

To see the output of the event log for a network adapter, !ndiskd provides a link to it in the State section of the **!ndiskd.netadapter** output. This is easier than the manual method of finding an event log's handle from a miniport block and using that to run the **!ndiskd.ndisevent** extension.

First, enter the **!ndiskd.netadapter** command with no parameters to see a list of network adapters and miniport drivers on the system. In the following example, look for the handle for the Marvell AVASTAR Wireless-AC Network Controller, ffffc804b9e6f1a0.

```
1: kd> !ndiskd.netadapter
    Driver             NetAdapter          Name
    ffffc804af2e3710   ffffc804b9e6f1a0    Marvell AVASTAR Wireless-AC Network Controller
    ffffc804b99b9020   ffffc804b9c301a0    WAN Miniport (Network Monitor)
    ffffc804b99b9020   ffffc804b9c2a1a0    WAN Miniport (IPv6)
    ffffc804b99b9020   ffffc804b8a8a1a0    WAN Miniport (IP)
    ffffc804ae9d7020   ffffc804b9ceb1a0    WAN Miniport (PPPOE)
    ffffc804b9ca5900   ffffc804b9e601a0    WAN Miniport (PPTP)
    ffffc804b99dc720   ffffc804b99b01a0    WAN Miniport (L2TP)
    ffffc804b86581b0   ffffc804b9c6c1a0    WAN Miniport (IKEv2)
    ffffc804ad4a7250   ffffc804b99651a0    WAN Miniport (SSTP)
    ffffc804b11c4020   ffffc804b85821a0    Microsoft ISATAP Adapter
    ffffc804b11c4020   ffffc804b71731a0    Microsoft ISATAP Adapter #2
    ffffc804ad725020   ffffc804b05e71a0    Surface Ethernet Adapter #2
    ffffc804b0bf0020   ffffc804b0c011a0    Bluetooth Device (Personal Area Network)
    ffffc804aef695e0   ffffc804aed331a0    TAP-Windows Adapter V9
```

Now, click on the link for that NetAdapter or enter the **!ndiskd.netadapter -handle** command to see its details. Look for the "Show state history" link to the right of the Device PnP field, in the State section.

```
1: kd> !ndiskd.netadapter ffffc804b9e6f1a0


MINIPORT

    Marvell AVASTAR Wireless-AC Network Controller

    Ndis handle        ffffc804b9e6f1a0
    Ndis API version   v6.50
    Adapter context    ffffc804af3b1100
    Driver             ffffc804af2e3710 - mrvlpcie8897  v1.0
    Network interface  ffffc804aea60a20

    Media type         802.3
    Physical medium    NdisPhysicalMediumUnspecified
    Device instance    PCI\VEN_11AB&DEV_2B38&SUBSYS_045E0001&REV_00\4&379f07b2&0&00E0
    Device object      ffffc804b9e6f050    More information
    MAC address        c0-33-5e-13-22-f7


STATE

    Miniport           INITIALIZING
    Device PnP         ADDED               Show state history
    Datapath           Normal
    Operational status DOWN
    Operational flags  [No flags set]
    Admin status       ADMIN_UP
    Media              MediaConnectUnknown
    Power              D0
    References         1                   Show detail
    Total resets       0
    Pending OID        None
    Flags              IN_INITIALIZE, NOT_BUS_MASTER, DEFAULT_PORT_ACTIVATED,
                       NOT_SUPPORTS_MEDIA_SENSE, DOES_LOOPBACK, MEDIA_CONNECTED
    PnP flags          PM_SUPPORTED, RECEIVED_START, HARDWARE_DEVICE


WDI

    This system supports WDI.
    Learn more about the associated WDI state


BINDINGS

    Protocol list      Driver              Open              Context
    No protocols are bound to this miniport

    Filter list        Driver              Module            Context
    No filters are bound to this miniport



MORE INFORMATION

    Driver handlers                        Task offloads
    Power management                       PM protocol offloads
    Pending OIDs                           Timers
    Pending NBLs                           Receive side throttling
    Wake-on-LAN (WoL)                      Packet filter
    Receive queues                         Receive filtering
    RSS                                    NIC switch
    Hardware resources                     Selective suspend
    NDIS ports                             WMI guids
    Diagnostic log
```

Now you can click the "Show state history" link or use the net adapter's handle to enter the **!ndiskd.netadapter -handle -log** command, which will show you the PnP event log for this miniport's miniport driver.

```
1: kd> !ndiskd.netadapter ffffc804b9e6f1a0 -log


MINIPORT PM & PNP EVENTS

    Event              Timestamp           (most recent event at bottom)
    DeviceAdded
                       13 ms later
    DeviceStart
                       Mon Mar 20 21:27:07.106 2017 (UTC - 7:00) Now?

    Set a breakpoint on the next event
```

See also

Network Driver Design Guide

Windows Vista and Later Networking Reference

Debugging the Network Stack

NDIS extensions (Ndiskd.dll)

!ndiskd.help

!ndiskd.netadapter

!ndiskd.ndisstack

10/23/2019 • 2 minutes to read • Edit Online

Note Third party network driver developers are not expected to manually use this extension command. You can run it to see the information it displays but you are not able to reuse the details it provides in your driver.

The **!ndiskd.ndisstack** extension displays a debug stack trace.

```
!ndiskd.ndisstack [-handle <x>] [-statistics]
```

Parameters

-handle
Required. Handle of the stack block.

-statistics
Shows debugging statistics.

DLL

Ndiskd.dll

See also

Network Driver Design Guide

Windows Vista and Later Networking Reference

Debugging the Network Stack

NDIS extensions (Ndiskd.dll)

!ndiskd.help

!ndiskd.wdiadapter

10/23/2019 • 2 minutes to read • Edit Online

The **!ndiskd.wdiadapter** extension displays information about a WDIWiFi!CAdapter structure. If you run this extension with no parameters, !ndiskd will display a list of all WDIWiFi!CAdapter structures.

For more information about WDI miniport drivers, see the WDI Miniport Driver Design Guide.

For more information about WDI miniport driver reference, see WDI Miniport Driver Reference.

```
!ndiskd.wdiadapter [-handle <x>] [-pm] [-rcvfilter]
```

Parameters

-handle
Handle of a CAdapter object.

-pm
Shows power management state and capabilities.

-rcvfilter
Shows receive filtering capabilities.

DLL

Ndiskd.dll

Examples

Run the **!ndiskd.wdiadapter** extension with no parameters to see a list of all CAdapter objects, along with details for each of their WDI Adapters. In the following example, there is only one CAdapter structure. The handle for its associated WDI Adapter is ffffc804af396000.

```
1: kd> !ndiskd.wdiadapter
    CAdapter
    ffffc804af396000 - WDI Adapter


WDI ADAPTER

    Object             ffffc804af396000
    Miniport           ffffc804b9e6f1a0
    WDI driver         ffffc804b8ce7c40 - WDI MiniDriver
    NetGuid            47cb3aa5-e29c-4628-80bd-5023c53fcccb
    Power              D0

    CCtlPlane          ffffc804af396080
    CTxMgr             ffffc804af399790


DEVICE COMMANDS

    Scheduler state    DeviceCommandSchedulerStateInit
    [No queued device commands found]

Device Command Scheduler


SERIALIZED JOBS

    Job                State              Type               Child
    ffffc804b9f33000   StepPending        WiFiJobMiniportInitialize



ACTIVE JOBS

    Job                State              Type               Child
    ffffc804b9f33000   StepPending        WiFiJobMiniportInitialize



EVENTS

    Processing?        No

    Event              Status             Type
    [No events found]

Event Queue


MORE INFORMATION

    Power management                      Receive filtering
```

Now you can either click on the "Power management" and "Receive filtering" links at the bottom of the CAdapter structure's description, or you can enter the **!ndiskd.wdiadapter -handle** command with either the *-pm* or *-rcvfilter* option. The following example shows output for the *-rcvfilter* option.

```
1: kd> !ndiskd.wdiadapter ffffc804af396000 -rcvfilter


RECEIVE FILTER

    Current Configuration

    Revision                            0
    Flags                               0
    Enabled filter types                [No flags set]
    Enabled queue types                 [No flags set]
    Max Queues                          0
    Queue properties                    [No flags set]
    Filter tests                        [No flags set]
    Header types                        [No flags set]
    MAC header fields                   [No flags set]
    Max MAC header filters              0
    Max queue groups                    0
    Max queues per group                0
    Min lookahead split size            0
    Max lookahead split size            0
```

See also

Network Driver Design Guide

Windows Vista and Later Networking Reference

Debugging the Network Stack

NDIS extensions (Ndiskd.dll)

!ndiskd.help

WDI Miniport Driver Design Guide

WDI Miniport Driver Reference

!ndiskd.wdiminidriver

10/23/2019 • 2 minutes to read • Edit Online

The **!ndiskd.wdiminidriver** extension displays information about one or more CMiniportDriver structures. If you run this extension with no parameters, !ndiskd will display a list of all CMiniportDriver structures.

For more information about WDI miniport drivers, see the WDI Miniport Driver Design Guide.

For more information about WDI miniport driver reference, see WDI Miniport Driver Reference.

```
!ndiskd.wdiminidriver [-handle <x>] [-pm] [-rcvfilter]
```

Parameters

-handle
Handle of a CMiniportDriver object.

-basic
Displays basic information about the miniport driver.

-handlers
Displays this driver's miniport handlers.

DLL

Ndiskd.dll

Examples

Run the **!ndiskd.wdiminidriver** extension with no parameters to see a list of all CMiniportDriver objects. In the following example, there is only one CMiniportDriver object. The handle of its WdiMiniDriver is ffffc804b8ce7c40.

```
1: kd> !ndiskd.wdiminidriver
    WdiMiniDriver      Name
    ffffc804b8ce7c40 - WDI MiniDriver
```

Click on the WdiMiniDriver's handle or enter the **!ndiskd.wdiminidriver -handle** command to see details for this WDI miniport driver.

```
1: kd> !ndiskd.wdiminidriver ffffc804b8ce7c40


WDI MINIPORT DRIVER

    Object             ffffc804b8ce7c40
    DRIVER_OBJECT      ffffc804b90a4ac0

    Ndis API version   v6.50
    NDIS driver        ffffc804af2e3710 - mrvlpcie8897
    Driver version     v1.0
    WDI API version    v1.0.1


    Handlers
```

Now you can click the "Handlers" link at the bottom of the WDI Miniport Driver's details, or you can use the handle to enter the **!ndiskd.wdiminidriver -handle -handlers** command, to see a list of all this WDI Miniport Driver's handlers.

```
1: kd> !ndiskd.wdiminidriver ffffc804b8ce7c40 -handlers


HANDLERS

    Miniport Handler                    Function pointer   Symbol (if available)
    InitializeHandlerEx                 [None]
    SetOptionsHandler                   fffff80965e8cae0   mrvlpcie8897!wlan_cmd_queue_deinit
    UnloadHandler                       fffff80965e71a08   mrvlpcie8897!MPUnload
    HaltHandlerEx                       [None]
    ShutdownHandlerEx                   fffff80965e71974   mrvlpcie8897!MPShutdown

    CheckForHangHandlerEx               [None]
    ResetHandlerEx                      [None]

    PauseHandler                        [None]
    RestartHandler                      [None]

    OidRequestHandler                   [None]
    SendNetBufferListsHandler           [None]
    ReturnNetBufferListsHandler         [None]
    CancelSendHandler                   [None]
    CancelOidRequestHandler             [None]
    DirectOidRequestHandler             [None]
    CancelDirectOidRequestHandler       [None]
    DevicePnPEventNotifyHandler         fffff80965e71868   mrvlpcie8897!MPPnPEventNotify

    AllocateAdapterHandler              fffff80965e713ec   mrvlpcie8897!MPAllocateTalAdapter
    FreeAdapterHandler                  fffff80965e717e8   mrvlpcie8897!MPFreeTalAdapter
    OpenAdapterHandler                  fffff80965e719b8   mrvlpcie8897!MPTaskOpenHandler
    CloseAdapterHandler                 fffff80965e719ac   mrvlpcie8897!MPTaskCloseHandler
    StartOperationHandler               [None]
    StopOperationHandler                [None]
    PostPauseHandler                    [None]
    PostRestartHandler                  [None]
    HangDiagnoseHandler                 fffff80965e715d8   mrvlpcie8897!MPDiagnosticHandler
    TalTxRxInitializeHandler            fffff80965e752d4   mrvlpcie8897!TalDataPathInitialize
    TalTxRxDeinitializeHandler          fffff80965e7527c   mrvlpcie8897!TalDataPathDeinitialize

    OpenAdapterCompleteHandler          fffff80965ffab50   wdiwifi!WDIOpenAdapterCompleteHandler
    CloseAdapterCompleteHandler         fffff80965fface0   wdiwifi!WDICloseAdapterCompleteHandler
```

See also

Network Driver Design Guide

Windows Vista and Later Networking Reference

Debugging the Network Stack

NDIS extensions (Ndiskd.dll)

!ndiskd.help

WDI Miniport Driver Design Guide

WDI Miniport Driver Reference

!ndiskd.nwadapter

10/23/2019 • 2 minutes to read • Edit Online

The **!ndiskd.nwadapter** extension displays information about one or more nwifi!ADAPT structures. If you run this extension with no parameters, !ndiskd will display a list of all nwifi!ADAPT structures.

```
!ndiskd.nwadapter [-handle <x>]
```

Parameters

-handle
Address of an ADAPT block.

DLL

Ndiskd.dll

See also

Network Driver Design Guide

Windows Vista and Later Networking Reference

Debugging the Network Stack

NDIS extensions (Ndiskd.dll)

!ndiskd.help

!ndiskd.ndisrwlock

10/23/2019 • 2 minutes to read • Edit Online

The **!ndiskd.ndisrwlock** extension displays information about an **NDIS_RW_LOCK_EX** lock structure.

```
!ndiskd.ndisrwlock [-handle <x>]
```

Parameters

-handle
Required. Handle of the lock structure.

DLL

Ndiskd.dll

Examples

Use the **!ndiskd.ndisrwlock** extension if you create your own RW lock and would want to inspect it. To obtain the handle for an RW lock, use the *poi* command to dereference the address of your driver's lock. The following snippet shows how to look at a lock that the TCIPIP protocol was using at the time of the example.

```
0: kd> !ndiskd.ndisrwlock poi(tcpip!gAleHashtableLock)


NDIS READ-WRITE LOCK

    Allocated by       [NDIS generic object]
    Exclusive access   Not acquired
    Read-only access   0 references

    Set a breakpoint on acquire/release
```

To observe the driver using this RW lock, click on the "Set a breakpoint on acquire/release" link at the bottom of the RW lock's details. After setting the breakpoint, enter the **g** command to let the debugee machine run and hit the breakpoint.

```
0: kd> ba r4 ffffe00bc3fc22f8
0: kd> g
Breakpoint 0 hit
nt!KeTestSpinLock+0x3:
fffff802`0d69eb53 4885c0          test    rax,rax
```

Now you can re-run the same **!ndiskd.ndisrwlock** command to see that this RW lock has one Read-only access reference.

```
0: kd> !ndiskd.ndisrwlock poi(tcpip!gAleHashtableLock)


NDIS READ-WRITE LOCK

    Allocated by       [NDIS generic object]
    Exclusive access   Not acquired
    Read-only access   1 reference

    Set a breakpoint on acquire/release
```

See also

Network Driver Design Guide

Windows Vista and Later Networking Reference

Debugging the Network Stack

NDIS extensions (Ndiskd.dll)

!ndiskd.help

NDIS_RW_LOCK_EX

!ndiskd.ndisslot

10/23/2019 • 2 minutes to read • Edit Online

Note Third party network driver developers are not expected to manually use this extension command. You can run it to see the information it displays but you are not able to reuse the details it provides in your driver.

The **!ndiskd.ndisslot** extension displays the contents of an NDIS per-processor variable. If you run this extension with no parameters, !ndiskd will display a list of all NDIS per-processor variables on the system.

```
!ndiskd.ndisslot [-handle <x>] [-itemtype <str>]
```

Parameters

-handle
Handle of the slot.

-itemtype
Type of the value stored in the slot.

DLL

Ndiskd.dll

Examples

Run the **!ndiskd.ndisslot** extension with no parameters to see a list of all per-processor slot variables. The following example output has excised the middle portion of the list for brevity.

```
1: kd> !ndiskd.ndisslot
    Per-processor slot                  Summary of contents
    ffffc804ae060000 - NDrw             All values are zero
    ffffc804ae060008 - NDrw             All values are zero
    ffffc804ae060010 - NDrw             All values are zero
    ffffc804ae060018 - NDrw             All values are zero
    ffffc804ae060020 - NDrw             All values are zero
    ffffc804ae060028 - NDrw             All values are zero
    ffffc804ae060030 - NDrw             All values are zero
    ffffc804ae060038 - NDrw             All values are zero
    ffffc804ae060040 - NDrw             All values are zero
    ffffc804ae060048 - NDrw             All values are zero

...

    ffffc804ae060910 - NDtk             All values are zero
    ffffc804ae060918 - NDtk             All values are zero
    ffffc804ae060920 - tsR              All values are non-zero
    ffffc804ae060928 - NDrw             All values are zero
    ffffc804ae060930 - NDtk             All values are zero
    ffffc804ae060938 - NDtk             All values are zero
    ffffc804ae060940 - NDtk             All values are zero
    ffffc804ae060948 - NDtk             All values are zero
    ffffc804ae060950 - NDrw             All values are zero
    ffffc804ae060958 - NDrw             All values are zero

    Statistics
    Descriptors        1
    Total slots        512
    Slots available    275
    Slots used         237
    Efficiency         46%
```

Clicking on one of the handles for the per-processor slot variables will show you the details for that variable. The following example uses the handle ffffc804ae060920 for the tsR variable, from the previous example.

```
1: kd> !ndiskd.ndisslot ffffc804ae060920
    Processor          Slot value
    00                 00000006
    01                 00000006
    02                 00000006
    03                 00000006
```

See also

Network Driver Design Guide

Windows Vista and Later Networking Reference

Debugging the Network Stack

NDIS extensions (Ndiskd.dll)

!ndiskd.help

!ndiskd.ndis

12/5/2018 • 2 minutes to read • Edit Online

The **!ndiskd.ndis** extension displays build information about ndis.sys.

```
!ndiskd.ndis
```

Parameters

This extension has no parameters.

DLL

Ndiskd.dll

Examples

The following example shows that the debugee machine has a Checked Ndis build.

```
0: kd> !ndiskd.ndis
Checked Ndis
```

!ndiskd.dbglevel

10/23/2019 • 2 minutes to read • Edit Online

The **!ndiskd.dbglevel** extension displays and optionally changes the current NDIS debug level.

Warning **!ndiskd.dbglevel** has been superceded by WPP (Windows software trace preprocessor) and Driver Verifier. !ndiskd will give you the following warning if your target system does not support **!ndiskd.dbglevel**.

```
0: kd> !ndiskd.dbglevel
    This target does not support tracing through !ndiskd.dbglevel or
    !ndiskd.dbgsystems.
    Learn how to collect traces with WPP
```

If you click on the link at the bottom of the warning, !ndiskd will give you more information.

```
0: kd> !ndiskd.help wpptracing
    WPP traces are fast, flexible, and detailed.  Plus, starting with Windows 8
    and Windows Server 2012, you can automatically decode NDIS traces using the
    symbol file.  Just point TraceView (or tracepdb.exe) at NDIS.PDB, and it
    will be able to get all the TMFs it needs to trace NDIS activity.

    If you would like traces to be printed in the debugger window, you use the
    !wmitrace extension.  For example, you might enable traces with this:

    !wmitrace.searchpath c:\path\to\TMF\files
    !wmitrace.start ndis -kd
    !wmitrace.enable ndis {DD7A21E6-A651-46D4-B7C2-66543067B869} -level 4 -flag 0x31f3
```

For more information about WPP, see WPP Software Tracing.

For more information about Driver Verifier, see Driver Verifier.

For more information about WMI tracing, see WMI Tracing Extensions (Wmitrace.dll).

```
!ndiskd.dbglevel [-level <str>]
```

Parameters

-level
The level of debugging verbosity. Possible values are:

- NONE - disables debug tracing
- FATAL - enables fatal errors to be printed
- ERROR - enables errors to be printed
- WARN - enables warnings to be printed
- INFO - enables informational messages to be printed
- VERBOSE - enables all debug traces to be printed

DLL

Ndiskd.dll

Remarks

This extension applies to checked NDIS.sys only. To check the build info of NDIS.sys, run the **!ndiskd.ndis** extension.

See also

Network Driver Design Guide

Windows Vista and Later Networking Reference

Debugging the Network Stack

NDIS extensions (Ndiskd.dll)

!ndiskd.help

!ndiskd.ndis

WPP Software Tracing

Driver Verifier

WMI Tracing Extensions (Wmitrace.dll)

!ndiskd.dbgsystems

10/23/2019 • 2 minutes to read • Edit Online

The **!ndiskd.dbgsystems** extension displays and optionally changes the NDIS subsystems that have debug traces enabled.

Warning **!ndiskd.dbgsystems** has been superceded by WPP (Windows software trace preprocessor) and Driver Verifier. !ndiskd will give you the following warning if your target system does not support **!ndiskd.dbgsystems**.

```
0: kd> !ndiskd.dbgsystems
    This target does not support tracing through !ndiskd.dbglevel or
    !ndiskd.dbgsystems.
    Learn how to collect traces with WPP
```

If you click on the link at the bottom of the warning, !ndiskd will give you more information.

```
0: kd> !ndiskd.help wpptracing
    WPP traces are fast, flexible, and detailed.  Plus, starting with Windows 8
    and Windows Server 2012, you can automatically decode NDIS traces using the
    symbol file.  Just point TraceView (or tracepdb.exe) at NDIS.PDB, and it
    will be able to get all the TMFs it needs to trace NDIS activity.

    If you would like traces to be printed in the debugger window, you use the
    !wmitrace extension.  For example, you might enable traces with this:

    !wmitrace.searchpath c:\path\to\TMF\files
    !wmitrace.start ndis -kd
    !wmitrace.enable ndis {DD7A21E6-A651-46D4-B7C2-66543067B869} -level 4 -flag 0x31f3
```

For more information about WPP, see WPP Software Tracing.

For more information about Driver Verifier, see Driver Verifier.

For more information about WMI tracing, see WMI Tracing Extensions (Wmitrace.dll).

```
!ndiskd.dbgsystems [-subsystem <any>]
```

Parameters

-subsystem
The subsystem to toggle.

If multiple components are selected, separate them with spaces. If a previously-selected component is repeated, its debug monitoring will be toggled off. The following values are possible:

Value	Meaning
INIT	Traces adapter initialization.

CONFIG	Traces adapter configuration.
SEND	Traces sending data over the network.
RECV	Traces receiving data from the network.
PROTOCOL	Traces protocol operations.
BIND	Traces binding operations.
BUS_QUERY	Traces bus queries.
REGISTRY	Traces registry operations.
MEMORY	Traces memory management.
FILTER	Traces filter operations.
REQUEST	Traces requests.
WORK_ITEM	Traces work-item operations.
PNP	Traces Plug and Play operations.
PM	Traces Power Management operations.
OPEN	Traces operations that open reference objects.
LOCKS	Traces locking operations.
RESET	Traces resetting operations.
WMI	Traces Windows Management Instrumentation operations.
NDIS_CO	Traces Connection-Oriented NDIS.
REFERENCE	Traces reference operations.

DLL

Remarks

This extension applies to checked NDIS.sys only. To check the build info of NDIS.sys, run the **!ndiskd.ndis** extension.

See also

Network Driver Design Guide

Windows Vista and Later Networking Reference

Debugging the Network Stack

NDIS extensions (Ndiskd.dll)

!ndiskd.help

!ndiskd.ndis

WPP Software Tracing

Driver Verifier

WMI Tracing Extensions (Wmitrace.dll)

!ndiskd.ndiskdversion

10/23/2019 • 2 minutes to read • Edit Online

The **!ndiskd.ndiskdversion** extension displays information about !ndiskd itself.

```
!ndiskd.ndiskdversion
```

Parameters

This extension has no parameters.

DLL

Ndiskd.dll

Examples

The following example shows output for **!ndiskd.ndiskdversion**.

```
1: kd> !ndiskd.ndiskdversion
    NDISKD Version     17.01.00 (NDISKD codename "All your WDF are belong to us")
    Build              Release
    Debugger CPU       AMD64
    NDIS symbols       Private             More info
    Hyperlinks (DML)   Enabled
    Unicode            Enabled
    Debug NDISKD       Enabled
```

See also

Network Driver Design Guide

Windows Vista and Later Networking Reference

Debugging the Network Stack

NDIS extensions (Ndiskd.dll)

!ndiskd.help

!ndiskd.netreport

10/23/2019 • 3 minutes to read • Edit Online

The **!ndiskd.netreport** extension generates a visual report of the entire network stack. The report **!ndiskd.netreport** generates is an HTML file, and it will give you a link to its location. The HTML file contains detailed information about the network stack, so if you need to share it for analysis you can email it instead of having to send a large crash dump file.

```
!ndiskd.netreport [-outputpath <str>] [-jsononly]
```

Parameters

-outputpath
Specifies where to write the report file.

-jsononly
Only writes the raw data, no HTML.

DLL

Ndiskd.dll

Examples

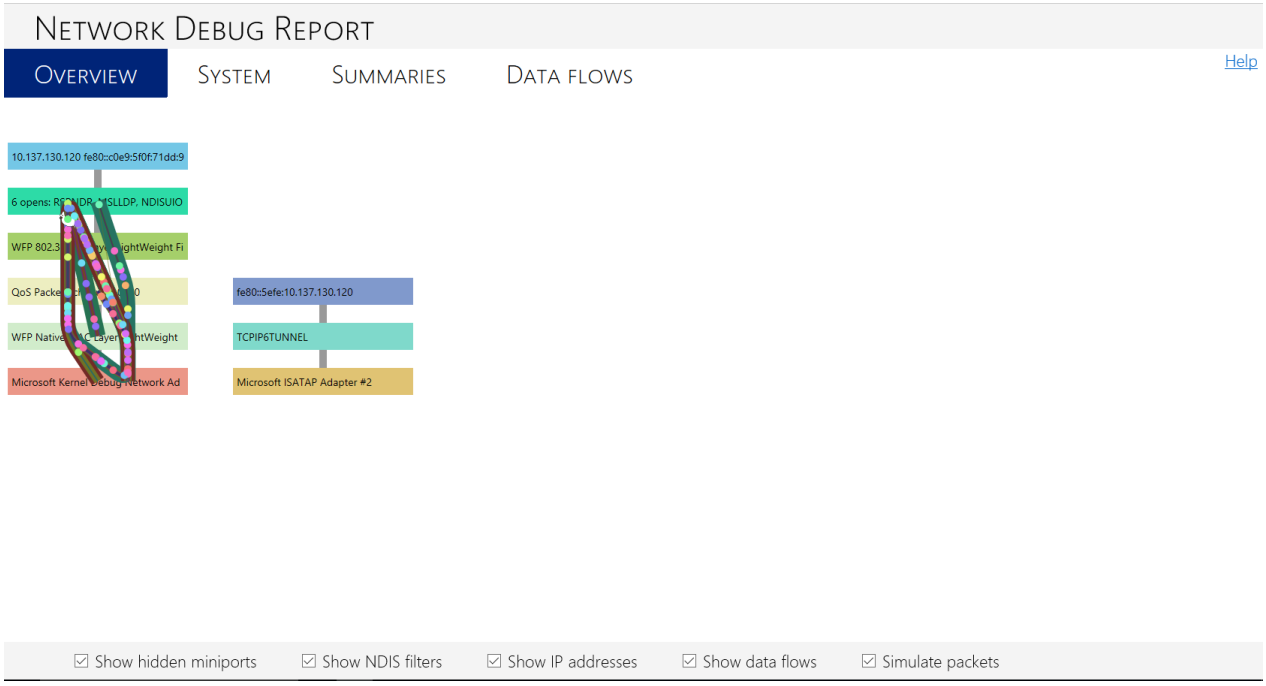
Run the **!ndiskd.netreport** extension to draw a box diagram of your network stack.

```
1: kd> !ndiskd.netreport


NETWORK STACK REPORT


    Want more stuff?  Rerun with the -verbose flag


    Report was saved to C:\Users\******\AppData\Local\Temp\NKDFE9F.html
    View the report                         Send in email
```

Click the "View the report" link at the bottom to see the report generated. The following image shows a net report generated from a crash dump file. Each vertical stack is a network adapter, broken down into layers showing the components of the stack. The color of each box is generated by hashing the name of the component, which means the same components will render with the same color every time you run the report. This means you can easily pick out a particular driver or adapter if you are debugging an issue with it.

As a comparison, the following image shows a net report generated from an active system instead of a crash dump file. Note that there are two more options at the bottom of the HTML page to "Show data flows" and "Simulate packets," and there is a fourth tab at the top of the report for "Data flows." These options appeared because the debugee machine had NBL tracking enabled, which lets **!ndiskd.netreport** parse the NBL tracking log to display the information visually. If NBL tracking is not turned on, these options will not appear. For more information about NBL tracking and the NBL log, see **!ndiskd.nbllog**.

By checking the "Show data flows" box, you can see the paths where the data is flowing. By checking the "Simulate packets" box, you can see animated circles moving up and down the data flow paths. Each circle represents a network packet.



This second example from an active system also shows another difference from the first example, which used a crash dump file. The target debugee machine in the second example was provisioned for kernel debugging over a network, so you can see the network adapter on the stack with the data flows is the Microsoft Kernel Debug Network Adapter. This adapter is usually hidden unless kernel debugging has been enabled on the debugee machine. In reality, the Kernel Debug Network Adapter has reserved the machine's Ethernet adapter for the debug session, so traffic is flowing over Ethernet.

The ability to visualize the network stack and see where traffic is flowing can enable you to quickly identify where a problem might be. This can be particularly helpful for virtual switches or servers, which have more complicated network diagrams than the previous examples. For example, on a Windows Server that uses NIC Teaming, you can see if multiple network stacks cross each other to balance the traffic load and identify if there is an issue at the bottom of one stack that is affecting another stack. To see an example of a network debug report that shows this, see Debugging the Network Stack. For more information about NIC Teaming, see Using NIC Teaming for Network Subsystem Performance.

!ndiskd.netreport also has other tabs at the top of the page for System, Summaries, and Data Flows (if applicable). These tabs contain further useful information about the state of the network stack. The following image shows the Network Interfaces tab, under the Summaries tab. The table in this tab lets you see more information about the names and identifiers for the network interfaces in the system.

NETWORK DEBUG REPORT

OVERVIEW SYSTEM SUMMARIES DATA FLOWS Help

NETWORK INTERFACES MAC ADDRESSES IPv4 & IPv6

NetLuid	IfIndex†	Guid	Name	H
24:00	1	aa87e4cd-7b53-11e6-b714-806e6f6e6963	Software Loopback Interface 1	ffffe00
06:8004	2	1fbfdb86-3a57-4162-aa1e-125281a854c7	WAN Miniport (Network Monitor)	ffffe00
23:8000	3	4090e55d-5787-47ea-af0b-41a355035edf	WAN Miniport (PPPOE)	ffffe00
131:8002	4	461fc1e5-d68f-43b4-9aef-dfbb7b6c4508	WAN Miniport (IKEv2)	ffffe00
06:8002	5	48312685-138a-4170-9369-e8439c3a1eeb	WAN Miniport (IP)	ffffe00
06:8000	6	551a82bc-0677-4fb9-a74b-f9cdba5f2b1e	Microsoft Kernel Debug Network Adapter	ffffe00
131:8000	7	6730c246-c98a-427e-9976-8b29dbc6f18a	Microsoft ISATAP Adapter	ffffe00
131:8001	8	6b204269-1a22-4fec-b672-cfa37a80f88d	WAN Miniport (SSTP)	ffffe00
131:8005	9	6d567b1a-2189-4288-b4ce-6ec580e8a570	Microsoft ISATAP Adapter #2	ffffe00
131:8004	10	755b22eb-a626-4575-b4f8-d2a4bc3eebb6	WAN Miniport (PPTP)	ffffe00
06:8003	11	7e573422-226f-4cb9-b43c-9e09934af8c5	WAN Miniport (IPv6)	ffffe00
131:8003	12	9bf86317-5c62-4fed-b177-50f723ab0f62	WAN Miniport (L2TP)	ffffe00
131:8006	13	bcd40482-26ac-41dd-b5c9-5cfa8a1510fe	Teredo Tunneling Pseudo-Interface	ffffe00
06:8001	14	cbddfde1-5570-4c65-9d47-52d63abce00c	Intel(R) 82579LM Gigabit Network Connection	ffffe00
06:00	15	fc2a0ae0-b103-11e6-b724-806e6f6e6963	Microsoft Kernel Debug Network Adapter-WFP Native MAC Layer LightWeight Filter-0000	ffffe00
06:01	16	fc2a0ae1-b103-11e6-b724-806e6f6e6963	Microsoft Kernel Debug Network Adapter-QoS Packet Scheduler-0000	ffffe00
06:02	17	fc2a0ae0-b103-11e6-b724-806e6f6e6963	Microsoft Kernel Debug Network Adapter-WFP 802.3 MAC Layer LightWeight Filter-0000	ffffe00

† Note that the IfIndex changes each time the network interface is started (or the computer is rebooted). Avoid using the IfIndex as a permanent identifier.

The Data flows tab, which appears if NBL tracking was enabled on the target system, shows a table of traffic events and details about each one. The following image shows the Data flows tab from the active system in the second example debug report described previously.

NETWORK DEBUG REPORT

OVERVIEW SYSTEM SUMMARIES DATA FLOWS Help

Occurances	Weighted	Failure?	Event	Context	Source	Destination
855	30.7%		Freed			
327	11.8%		MiniportReceived	ffffe00bc53ae181		ffffe00bc5ac5c70
			FilterReceived	00000020		
			IndicatedToProtocol_Synchronous	ffffe00bc30abbb1		ffffe00bc30abc10
			ProtocolReturned_Synchronous	ffffe00bc53ae181		ffffe00bc5ac5c70
			IndicatedToProtocol	ffffe00bc5813971		ffffe00bc5ac0010
			ProtocolReturned	ffffe00bc53ae181		ffffe00bc5ac5c70
			FilterReturned	ffffe00bc51a7971		ffffe00bc3f701a0
314	11.3%		Allocated			
			ProtocolSent	ffffe00bc4af2861	ffffe00bc5ab7c10	ffffe00bc5ac4880
			FilterSent	ffffe00bc53ae181		ffffe00bc5ac5c70
			FilterSent	ffffe00bc51a7971		ffffe00bc3f701a0
			SentToMiniport	ffffe00bc51a7971		ffffe00bc3f701a0
			MiniportSendCompleted	ffffe00bc53ae181		ffffe00bc5ac5c70
			FilterSendCompleted	ffffe00bc4af2861		ffffe00bc5ac4880
			FilterSendCompleted	00000030		
			SendCompleted	ffffe00bc5ab88a1		ffffe00bc5ab7c10
297	10.7%		MiniportReceived	ffffe00bc53ae181	ffffe00bc3f701a0	ffffe00bc5ac5c70
			FilterReceived	00000020		
			IndicatedToProtocol_Synchronous	ffffe00bc30abbb1		ffffe00bc30abc10
			ProtocolReturned_Synchronous	ffffe00bc53ae181		ffffe00bc5ac5c70
			IndicatedToProtocol	ffffe00bc5813971		ffffe00bc5ac0010
			ProtocolReturned	ffffe00bc53ae181		ffffe00bc5ac5c70
			FilterReturned	ffffe00bc51a7971		ffffe00bc3f701a0

See also

Network Driver Design Guide

Windows Vista and Later Networking Reference

Debugging the Network Stack

NDIS extensions (Ndiskd.dll)

!ndiskd.help

!ndiskd.nbllog

Using NIC Teaming for Network Subsystem Performance

RPC Extensions (Rpcexts.dll)

12/5/2018 • 2 minutes to read • Edit Online

Extension commands that are useful for debugging Microsoft Remote Procedure Call (RPC) can be found in Rpcexts.dll.

The Windows 2000 version of this extension DLL appears in the w2kfre and w2kchk directories. The Windows XP and later version of this extension DLL appear in the winxp directory.

For more information about how to use these extensions, see Using the RPC Debugger Extensions.

!rpcexts.checkrpcsym

12/5/2018 • 2 minutes to read • Edit Online

The **!rpcexts.checkrpcsym** extension command is obsolete; the output from the debugger should be sufficient to diagnose any RPC symbol problems.

!rpcexts.eeinfo

12/5/2018 • 2 minutes to read • Edit Online

The **!rpcexts.eeinfo** extension displays the extended error information chain.

```
!rpcexts.eeinfo EEInfoAddress
```

Parameters

EEInfoAddress
Specifies the address of the extended error information.

DLL

Windows 2000	Unavailable
Windows XP and later	Rpcexts.dll

Additional Information

For more information about debugging Microsoft Remote Procedure Call (RPC), see RPC Debugging.

Remarks

This extension displays the contents of all records in the extended error information chain.

The records are displayed in order, with the most recent records first. The records are separated by a line of dashes.

Here is an example (in which there is only one record):

```
0:001> !rpcexts.eeinfo 0xb015f0
Computer Name: (null)
ProcessID: 708 (0x2C4)
System Time is: 3/21/2000 4:3:0:264
Generating component: 8
Status: 14
Detection Location: 311
Flags:
Parameter 0:(Long value) : -30976 (0xFFFF8700)
Parameter 1:(Long value) : 16777343 (0x100007F)
```

If the chain is very long and you wish to see only one record, use **!rpcexts.eerecord** instead.

!rpcexts.eerecord

12/5/2018 • 2 minutes to read • Edit Online

The **!rpcexts.eerecord** extension displays the contents of an extended error information record.

```
!rpcexts.eerecord EERecordAddress
```

Parameters

EERecordAddress
Specifies the address of the extended error record.

DLL

Windows 2000	Unavailable
Windows XP and later	Rpcexts.dll

Additional Information

For more information about debugging Microsoft Remote Procedure Call (RPC), see RPC Debugging.

Remarks

This extension displays the contents of one extended error information record in the debugger. In most cases, it is easier to use **!rpcexts.eeinfo**, which displays the whole chain. If the chain is very long and you wish to see only one record, use **!eerecord** instead.

Here is an example:

```
0:001> !rpcexts.eerecord 0xb015f0
Computer Name: (null)
ProcessID: 708 (0x2C4)
System Time is: 3/21/2000 4:3:0:264
Generating component: 8
Status: 14
Detection Location: 311
Flags:
Parameter 0:(Long value) : -30976 (0xFFFF8700)
Parameter 1:(Long value) : 16777343 (0x100007F)
```

!rpcexts.getcallinfo

12/5/2018 • 3 minutes to read • Edit Online

The **!rpcexts.getcallinfo** extension searches the system's RPC state information for server-side call (SCALL) information.

```
!rpcexts.getcallinfo [ CallID | 0 [ IfStart | 0 [ ProcNum | 0xFFFF [ProcessID|0] ] ] ]
!rpcexts.getcallinfo -?
```

Parameters

CallID
Specifies the call ID. This parameter is optional; include it if you only want to display calls matching a specific *CallID* value.

IfStart
Specifies the first DWORD of the interface UUID on which the call was made. This parameter is optional; include it if you only want to display calls matching a specific *IfStart* value.

ProcNum
Specifies the procedure number of this call. (The RPC Run-Time identifies individual routines from an interface by numbering them by position in the IDL file -- the first routine in the interface is 0, the second 1, and so on.)

ProcessID
Specifies the process ID (PID) of the server process that owns the calls you want to display. This parameter is optional; omit it if you want to display calls owned by multiple processes.

-?
Displays some brief Help text for this extension in the Command Prompt window.

DLL

Windows 2000	Unavailable
Windows XP and later	Rpcexts.dll

Additional Information

For more information about debugging Microsoft Remote Procedure Call (RPC), see RPC Debugging.

Remarks

This extension can only be used with CDB or with user-mode WinDbg.

The parameters are parsed from left to right. To skip a parameter, supply the value 0. There is one exception to this rule: the *ProcNum* parameter is skipped by supplying the value 0xFFFF.

Here is an example:

```
0:002> !rpcexts.getcallinfo
Searching for call info ...
## PID  CELL ID   ST PNO IFSTART  TIDNUMBER CALLFLAG CALLID   LASTTIME CONN/CLN
-------------------------------------------------------------------------------
00c4 0000.0006 00 009 367abb81 0000.0007 00000001 0000004a 00018b41 0000.0005
00c4 0000.000a 00 007 367abb81 0000.002d 00000001 0000009f 000134ff 0000.0009
00c4 0000.000d 00 00f 82273fdc 0000.002d 00000001 00000002 00036cd8 0000.0042
00c4 0000.0010 00 00f 367abb81 0000.002d 00000001 00000078 00011636 0000.000f
00c4 0000.0012 00 00d 8d9f4e40 0000.0007 00000001 0000004f 000097bd 0000.0011
00c4 0000.0015 00 000 367abb81 0000.0004 00000001 0000004c 0002cccf 0000.0014
00c4 0000.0017 00 007 367abb81 0000.0004 00000001 00000006 0000cf5e 0000.0016
00c4 0000.0018 00 000 367abb81 0000.002d 00000001 0000000b 0001236f 0000.002a
00c4 0000.0019 01 00b 82273fdc 0000.0002 00000009 00000000 00018b19 00d0.0104
00c4 0000.001b 00 009 65a93890 0000.0007 00000001 000000ea 0003cd14 0000.001a
00c4 0000.0021 00 03b 8d9f4e40 0000.0013 00000001 0000000b 0001162c 0000.0020
00c4 0000.0022 01 008 82273fdc 0000.001f 00000009 00000000 00013405 00c4.02e8
00c4 0000.0024 00 007 367abb81 0000.0004 00000001 00000006 0000f198 0000.0023
00c4 0000.0026 00 000 367abb81 0000.0036 00000001 000000ab 00038049 0000.0025
00c4 0000.0027 01 00b 82273fdc 0000.001f 00000009 00000000 00020b7c 00a8.0228
00c4 0000.0028 01 008 82273fdc 0000.003e 00000009 00000000 0003a949 0294.02f0
00c4 0000.0029 00 00d 8d9f4e40 0000.002d 00000001 0000033f 0003831a 0000.0031
00c4 0000.0030 00 03b 8d9f4e40 0000.0013 00000001 00000002 00024e43 0000.002f
00c4 0000.0032 01 008 82273fdc 0000.001f 00000009 00000000 000118f3 022c.019c
00c4 0000.0035 00 007 367abb81 0000.0033 00000001 00000074 0001042d 0000.0034
00c4 0000.0038 00 007 367abb81 0000.002d 00000001 0000000a 0002a3e4 0000.0037
00c4 0000.003a 00 007 367abb81 0000.0036 00000001 00000063 0003b7b8 0000.0039
00c4 0000.003b 00 004 3ba0ffc0 0000.002d 00000001 00000005 0002dd79 0000.002e
00c4 0000.003f 01 008 82273fdc 0000.0002 00000009 00000000 000245c6 01c0.037c
00c4 0000.0043 01 008 82273fdc 0000.0002 00000009 00000000 00037d50 020c.0394
00c4 0000.0049 00 008 8d9f4e40 0000.0007 00000001 000002b1 0004e900 0000.0048
0170 0000.0009 01 002 e60c73e6 0000.0002 00000009 baadf00d 0004ad30 020c.03a4
0170 0000.000a 01 002 0b0a6584 0000.0008 00000009 baadf00d 0001187b 00c4.012c
0170 0000.000c 01 002 0b0a6584 0000.0008 00000009 baadf00d 00011cdc 022c.019c
0170 0000.000d 01 003 00000136 0000.0011 00000009 baadf00d 00034845 020c.02b4
0170 0000.000e 01 000 412f241e 0000.0002 00000009 baadf00d 00012491 0294.02b8
0170 0000.000f 01 002 0b0a6584 0000.0011 00000009 baadf00d 000492e7 026c.0118
0170 0000.0010 01 002 e60c73e6 0000.0013 00000009 baadf00d 0004ab78 0378.038c
0170 0000.0014 01 004 e60c73e6 0000.0011 00000001 baadf00d 0002bc25 0378.024c
0170 0000.0015 01 003 00000136 0000.0013 00000009 00000003 00031d8d 0378.00b8
0170 0000.0018 01 004 00000136 0000.0002 00000001 baadf00d 00032e05 020c.026c
020c 0000.0004 01 003 00000132 0000.000b 00000009 00000000 00034953 0170.0240
020c 0000.000e 01 001 2f5f6520 0000.001e 00000009 00120006 00035bac 020c.03b4
020c 0000.0010 01 000 629b9f66 0000.000f 00000009 00000000 000279ff 00a8.0194
020c 0000.0011 01 004 faedcf59 0000.0003 00000009 00000012 0003836b 0378.024c
020c 0000.0012 01 001 629b9f66 0000.000f 00000009 00000000 0003657e 020c.02ec
020c 0000.0017 01 005 00000134 0000.0002 00000001 00000016 0003836b 0378.024c
020c 0000.001d 01 001 2f5f6520 0000.0014 00000001 0020007d 000351b2 020c.0258
0294 0000.0004 01 004 00000132 0000.0002 00000009 00000000 0003b786 0170.01ac
0378 0000.0004 01 003 00000134 0000.0003 0000000b 00300038 0002d896 020c.021c
026c 0000.0004 02 000 19bb5061 0000.0002 00000001 00000001 0004caa5 0000.0003
```

For a similar example using the DbgRpc tool, see Get RPC Call Information.

!rpcexts.getclientcallinfo

12/5/2018 • 2 minutes to read • Edit Online

The **!rpcexts.getclientcallinfo** extension searches the system's RPC state information for client call (CCALL) information.

```
!rpcexts.getclientcallinfo [ CallID | 0 [ IfStart | 0 [ ProcNum | 0xFFFF [ProcessID|0] ] ] ]
!rpcexts.getclientcallinfo -?
```

Parameters

CallID
Specifies the call ID. This parameter is optional; include it if you only want to display calls matching a specific *CallID* value.

IfStart
Specifies the first DWORD of the interface UUID on which the call was made. This parameter is optional; include it if you only want to display calls matching a specific *IfStart* value.

ProcNum
Specifies the procedure number of this call. (The RPC Run-Time identifies individual routines from an interface by numbering them by position in the IDL file -- the first routine in the interface is 0, the second 1, and so on.) This parameter is optional; include it if you only want to display calls matching a specific *ProcNum* value.

ProcessID
Specifies the process ID (PID) of the client process that owns the calls you want to display. This parameter is optional; omit it if you want to display calls owned by multiple processes.

-?
Displays some brief Help text for this extension in the Command Prompt window.

DLL

Windows 2000	Unavailable
Windows XP and later	Rpcexts.dll

Additional Information

For more information about debugging Microsoft Remote Procedure Call (RPC), see RPC Debugging.

Remarks

This extension can only be used with CDB or with user-mode WinDbg. It is only available if full RPC state information is being gathered.

Here is an example:

```
0:002> !rpcexts.getclientcallinfo
Searching for call info ...
## PID  CELL ID   PNO  IFSTART  TIDNUMBER CALLID   LASTTIME PS CLTNUMBER ENDPOINT
-------------------------------------------------------------------------------
03d4 0000.0001 0000 19bb5061 0000.0000 00000001 0004ca9b 07 0000.0002 1118
```

For a similar example using the DbgRpc tool, see Get RPC Client Call Information.

!rpcexts.getdbgcell

12/5/2018 • 2 minutes to read • Edit Online

The **!rpcexts.getdbgcell** extension displays RPC state information for the specified cell.

```
!rpcexts.getdbgcell ProcessID CellID1.CellID2
!rpcexts.getdbgcell -?
```

Parameters

ProcessID

Specifies the process ID (PID) of the process whose server contains the desired cell.

CellID1.CellID2

Specifies the number of the cell to be displayed.

-?

Displays some brief Help text for this extension in the Command Prompt window.

DLL

Windows 2000	Unavailable
Windows XP and later	Rpcexts.dll

Additional Information

For more information about debugging Microsoft Remote Procedure Call (RPC), see RPC Debugging.

Remarks

This extension can only be used with CDB or with user-mode WinDbg.

Here is an example:

```
0:002> !rpcexts.getdbgcell c4 0.19
Getting cell info ...
Call
Status: Active
Procedure Number: 11
Interface UUID start (first DWORD only): 82273FDC
Call ID: 0x0 (0)
Servicing thread identifier: 0x0.3E
Call Flags: cached, LRPC
Last update time (in seconds since boot):1453.459 (0x5AD.1CB)
Caller (PID/TID) is: d0.1ac (208.428)
```

For a similar example using the DbgRpc tool, see Get RPC Cell Information.

!rpcexts.getendpointinfo

12/5/2018 • 2 minutes to read • Edit Online

The **!rpcexts.getendpointinfo** extension searches the system's RPC state information for endpoint information.

```
    !rpcexts.getendpointinfo [EndpointName]
!rpcexts.getendpointinfo -?
```

Parameters

EndpointName
Specifies the number of the endpoint to be displayed. If omitted, the endpoints for all processes on the system are displayed.

-?
Displays some brief Help text for this extension in the Command Prompt window.

DLL

Windows 2000	Unavailable
Windows XP and later	Rpcexts.dll

Additional Information

For more information about debugging Microsoft Remote Procedure Call (RPC), see RPC Debugging.

Remarks

This extension can only be used with CDB or with user-mode WinDbg.

Here is an example:

```
0:002> !rpcexts.getendpointinfo
Searching for endpoint info ...
## PID  CELL ID   ST PROTSEQ        ENDPOINT
------------------------------------------------------------
00a8 0000.0001 01            NMP \PIPE\InitShutdown
00a8 0000.0003 01            NMP \PIPE\SfcApi
00a8 0000.0004 01            NMP \PIPE\ProfMapApi
00a8 0000.0005 01           LRPC OLE5
00a8 0000.0007 01            NMP \pipe\winlogonrpc
00c4 0000.0001 01           LRPC ntsvcs
00c4 0000.0003 01            NMP \PIPE\ntsvcs
00c4 0000.0008 01            NMP \PIPE\scerpc
00d0 0000.0001 01            NMP \PIPE\lsass
00d0 0000.0003 01            NMP \pipe\WMIEP_d0
00d0 0000.0006 01           LRPC policyagent
00d0 0000.0007 01            NMP \PIPE\POLICYAGENT
0170 0000.0001 01           LRPC epmapper
0170 0000.0003 01            TCP 135
0170 0000.0005 01            SPX 34280
0170 0000.0006 01             NB 135
0170 0000.0007 01             NB 135
0170 0000.000b 01            NMP \pipe\epmapper
01c0 0000.0001 01            NMP \pipe\spoolss
01c0 0000.0003 01           LRPC spoolss
01c0 0000.0007 01           LRPC OLE7
020c 0000.0001 01           LRPC OLE2
020c 0000.0005 01           LRPC senssvc
020c 0000.0007 01            NMP \pipe\tapsrv
020c 0000.0009 01           LRPC tapsrvlpc
020c 0000.000c 01            NMP \PIPE\ROUTER
020c 0000.0016 01            NMP \pipe\WMIEP_20c
0218 0000.0001 01            NMP \PIPE\winreg
022c 0000.0001 01           LRPC LRPC0000022c.00000001
022c 0000.0003 01            TCP 1041
022c 0000.0005 01            SPX 24576
022c 0000.0006 01            NMP \PIPE\atsvc
0294 0000.0001 01           LRPC OLE3
0378 0000.0001 01           LRPC OLE9
026c 0000.0001 01            TCP 1118
0344 0000.0001 01           LRPC OLE12
```

For a similar example using the DbgRpc tool, see Get RPC Endpoint Information.

!rpcexts.getthreadinfo

12/5/2018 • 2 minutes to read • Edit Online

The **!rpcexts.getthreadinfo** extension searches the system's RPC state information for thread information.

```
!rpcexts.getthreadinfo ProcessID [ThreadID]
!rpcexts.getthreadinfo -?
```

Parameters

ProcessID
Specifies the process ID (PID) of the process containing the desired thread.

ThreadID
Specifies the thread ID of the thread to be displayed. If omitted, all threads in the specified process will be displayed.

-?
Displays some brief Help text for this extension in the Command Prompt window.

DLL

Windows 2000	Unavailable
Windows XP and later	Rpcexts.dll

Additional Information

For more information about debugging Microsoft Remote Procedure Call (RPC), see RPC Debugging.

Remarks

This extension can only be used with CDB or with user-mode WinDbg.

Here is an example:

```
0:002> !rpcexts.getthreadinfo 26c
Searching for thread info ...
## PID  CELL ID   ST TID      LASTTIME
-----------------------------------
026c 0000.0002 01 000003c4 0004caa5
026c 0000.0005 03 00000254 0004ca9b
```

For a similar example using the DbgRpc tool, see Get RPC Thread Information.

!rpcexts.help

12/5/2018 • 2 minutes to read • Edit Online

The **!rpcexts.help** extension displays a Help text in the Command Prompt window showing all Rpcexts.dll extension commands.

```
!rpcexts.help
```

DLL

Windows 2000	Rpcexts.dll
Windows XP and later	Rpcexts.dll

!rpcexts.rpcreadstack

12/5/2018 • 2 minutes to read • Edit Online

The **!rpcexts.rpcreadstack** extension reads an RPC client-side stack and retrieves the call information.

```
!rpcexts.rpcreadstack ThreadStackPointer
```

Parameters

ThreadStackPointer
Specifies the pointer to the thread stack.

DLL

Windows 2000	Unavailable
Windows XP and later	Rpcexts.dll

Additional Information

For more information about debugging Microsoft Remote Procedure Call (RPC), see RPC Debugging.

Remarks

For a common use of this extension, see Analyzing a Stuck Call Problem.

Here is an example:

```
0:001> !rpcexts.rpcreadstack 68fba4
CallID: 1
IfStart: 19bb5061
ProcNum: 0
        Protocol Sequence:      "ncacn_ip_tcp"  (Address: 00692ED8)
        NetworkAddress: ""      (Address: 00692F38)
        Endpoint:       "1120"  (Address: 00693988)
```

!rpcexts.rpctime

12/5/2018 • 2 minutes to read • Edit Online

The **!rpcexts.rpctime** extension displays the current system time.

```
!rpcexts.rpctime
```

DLL

Windows 2000	Rpcexts.dll
Windows XP and later	Rpcexts.dll

Remarks

This extension can only be used with CDB or with user-mode WinDbg.

Here is an example:

```
0:001> !rpcexts.rpctime
Current time is: 059931.126  (0x00ea1b.07e)
```

!rpcexts.thread

12/5/2018 • 2 minutes to read • Edit Online

The **!rpcexts.thread** extension displays the per-thread RPC information.

This extension command should not be confused with the **.thread (Set Register Context)** command or the **!thread** (!kdextx86.thread and !kdexts.thread) extension.

```
!rpcexts.thread TEB
```

Parameters

TEB
Specifies the address of the thread environment block (TEB).

DLL

Windows 2000	Rpcexts.dll
Windows XP and later	Rpcexts.dll

Additional Information

For more information about debugging Microsoft Remote Procedure Call (RPC), see RPC Debugging.

Remarks

This extension displays the per-thread RPC information. A field in the per-thread RPC information is the extended error information for this thread.

Here is an example:

```
0:001> !rpcexts.thread 7ffdd000
RPC TLS at 692e70

HandleToThread - 0x6c
SavedProcedure - 0x0
SavedParameter - 0x0
ActiveCall - 0x0
Context - 0x0
CancelTimeout - 0xffffffff
SecurityContext - 0x0
ExtendedStatus - 0x0
ThreadEEInfo - 0xb015f0
ThreadEvent at - 0x00692E78
fCallCancelled - 0x0
buffer cache array at - 0x00692E84
fAsync - 0x0
```

ACPI Extensions (Acpikd.dll and Kdexts.dll)

1/3/2019 • 2 minutes to read • Edit Online

Extension commands that are useful for debugging ACPI (Advanced Configuration and Power Interface) BIOS code can be found in Acpikd.dll and Kdexts.dll.

Some of the ACPI debugging extensions can be found in Winxp\Acpikd.dll, while others can be found in Winxp\Kdexts.dll.

!acpicache

12/5/2018 • 2 minutes to read • Edit Online

The **!acpicache** extension displays all of the Advanced Configuration and Power Interface (ACPI) tables cached by the HAL.

```
!acpicache [DisplayLevel]
```

Parameters

DisplayLevel
Specifies the detail level of the display. This value is either 0 for an abbreviated display or 1 for a more detailed display. The default value is 0.

DLL

Windows 2000	Unavailable
Windows XP and later	Kdexts.dll

Additional Information

For information about the ACPI, see the Microsoft Windows Driver Kit (WDK) documentation, the Windows SDK documentation, and *Microsoft Windows Internals* by Mark Russinovich and David Solomon. (These books and resources may not be available in some languages and countries.) Also see ACPI Debugging for information about other extensions that are associated with the ACPI.

!acpiinf

12/5/2018 • 2 minutes to read • Edit Online

The **!acpiinf** extension displays information on the configuration of the Advanced Configuration and Power Interface (ACPI), including the location of system tables and the contents of the ACPI fixed feature hardware.

!acpiinf

DLL

Windows 2000	Unavailable
Windows XP and later	Kdexts.dll

Additional Information

For information about the ACPI, see the Microsoft Windows Driver Kit (WDK) documentation, the Windows SDK documentation, and *Microsoft Windows Internals* by Mark Russinovich and David Solomon. (These books and resources may not be available in some languages and countries.) Also see ACPI Debugging for information about other extensions that are associated with the ACPI.

!acpiirqarb

1/3/2019 • 2 minutes to read • Edit Online

The **!acpiirqarb** extension displays the contents of the Advanced Configuration and Power Interface (ACPI) IRQ arbiter structure, which contains the configuration of I/O devices to system interrupt controller inputs and processor interrupt dispatch table (IDT) entries.

```
!acpiirqarb
```

DLL

Kdexts.dll

Additional Information

For information about the ACPI, see the Microsoft Windows Driver Kit (WDK) documentation, the Windows SDK documentation, and *Microsoft Windows Internals* by Mark Russinovich and David Solomon. (These books and resources may not be available in some languages and countries.) Also see ACPI Debugging for information about other extensions that are associated with the ACPI.

!acpikd.help

12/5/2018 • 2 minutes to read • Edit Online

The **!acpikd.help** extension displays a Help text in the Debugger Command window showing all Acpikd.dll extension commands.

```
!acpikd.help
```

DLL

Windows 2000	Acpikd.dll
Windows XP and later	Acpikd.dll

Additional Information

For more information, see ACPI Debugging.

!amli ?

12/5/2018 • 2 minutes to read • Edit Online

The **!amli ?** extension displays some Help text in the Debugger Command window for the **!amli** extension commands.

Syntax

```
!amli ? [Command]
```

Parameters

Command
Specifies the **!amli** command whose help is to be displayed. For example, **!amli ? set** displays help for the **!amli set** command. If *Command* is omitted, a list of all commands is displayed.

DLL

Kdexts.dll

Additional Information

For information about related commands and their uses, see The AMLI Debugger.

!amli bc

10/8/2019 • 2 minutes to read • Edit Online

The **!amli bc** extension permanently clears an AML breakpoint.

Syntax

```
!amli bc Breakpoint
!amli bc *
```

Parameters

Breakpoint Specifies the number of the breakpoint to be cleared.

***** Specifies that all breakpoints should be cleared.

DLL

Kdexts.dll

Additional Information

For information about related commands and their uses, see The AMLI Debugger.

Remarks

To determine the breakpoint number of a breakpoint, use the **!amli bl** extension.

!amli bd

10/8/2019 • 2 minutes to read • Edit Online

The **!amli bd** extension temporarily disables an AML breakpoint.

Syntax

```
!amli bd Breakpoint!amli bd *
```

Parameters

Breakpoint Specifies the number of the breakpoint to be disabled.

***** Specifies that all breakpoints should be disabled.

DLL

Kdexts.dll

Additional Information

For information about related commands and their uses, see The AMLI Debugger.

Remarks

A disabled breakpoint can be re-enabled by using the **!amli be** extension.

To determine the breakpoint number of a breakpoint, use the **!amli bl** extension.

Here is an example of this command:

```
kd> !amli bl
 0:  c29accf5 [\_WAK]
 1:  c29c20a5 [\_SB.PCI0.ISA.BAT1._BST]

kd> !amli bd 1
```

!amli be

10/8/2019 • 2 minutes to read • Edit Online

The **!amli be** extension enables an AML breakpoint.

Syntax

```
!amli be Breakpoint!amli be *
```

Parameters

Breakpoint Specifies the breakpoint number of the breakpoint to be enabled.

***** Specifies that all breakpoints should be enabled.

DLL

Kdexts.dll

Additional Information

For information about related commands and their uses, see The AMLI Debugger.

Remarks

All breakpoints are enabled when they are created. Breakpoints are only disabled if you have used the **!amli bd** extension.

To determine the breakpoint number of a breakpoint, use the **!amli bl** extension.

!amli bl

12/5/2018 • 2 minutes to read • Edit Online

The **!amli bl** extension displays a list of all AML breakpoints.

Syntax

```
!amli bl
```

DLL

Kdexts.dll

Additional Information

For information about related commands and their uses, see The AMLI Debugger.

Remarks

The AMLI Debugger supports a maximum of ten breakpoints.

Here is an example of the **!amli bl** extension:

```
kd> !amli bl
 0: <e> ffffffff80e5e2f1:[\_SB.LNKD._SRS]
 1: <e> ffffffff80e5d969:[\_SB.LNKB._STA]
 2: <d> ffffffff80e630c9:[\_WAK]
 3: <e> ffffffff80e612c9:[\_SB.MBRD._CRS]
```

The first column gives the breakpoint number. The **<e>** and **<d>** marks indicate whether the breakpoint is enabled or disabled. The address of the breakpoint is in the next column. Finally, the method containing the breakpoint is listed, with the offset of the breakpoint if it is not set at the start of the method.

!amli bp

12/5/2018 • 2 minutes to read • Edit Online

The **!amli bp** extension places a breakpoint in AML code.

Syntax

```
!amli bp { MethodName | CodeAddress }
```

Parameters

MethodName
Specifies the full path of the method name on which the breakpoint will be set.

CodeAddress
Specifies the address of the AML code at which the breakpoint will be set. If *CodeAddress* is prefixed with two percent signs (**%%**), it is interpreted as a physical address. Otherwise, it is interpreted as a virtual address.

DLL

Kdexts.dll

Additional Information

For information about related commands and their uses, see The AMLI Debugger.

Remarks

The AMLI Debugger supports a maximum of 10 breakpoints.

Here is an example. The following command will set a breakpoint on the _DCK method:

```
kd> !amli bp \_sb.pci0.dock._dck
```

!amli cl

12/5/2018 • 2 minutes to read • Edit Online

The **!amli cl** extension clears the AML interpreter's event log.

Syntax

```
!amli cl
```

DLL

Kdexts.dll

Additional Information

For information about related commands and their uses, see The AMLI Debugger.

!amli debugger

12/5/2018 • 2 minutes to read • Edit Online

The **!amli debugger** extension breaks into the AMLI Debugger.

Syntax

```
!amli debugger
```

DLL

Kdexts.dll

Additional Information

For information about related commands and their uses, see The AMLI Debugger.

Remarks

When this command is issued, notification is sent to the AML interpreter. The next time the interpreter is active, it will immediately break into the AMLI Debugger.

The **!amli debugger** extension only causes one break. If you want it to break again, you need to use this extension again, or set a breakpoint.

!amli dh

12/5/2018 • 2 minutes to read • Edit Online

The **!amli dh** extension displays the AML interpreter's internal heap block.

Syntax

```
!amli dh [HeapAddress]
```

Parameters

HeapAddress
Specifies the address of the heap block. If this is omitted, the global heap is displayed.

DLL

Kdexts.dll

Additional Information

For information about related commands and their uses, see The AMLI Debugger.

!amli dl

12/5/2018 • 2 minutes to read • Edit Online

The **!amli dl** extension displays a portion of the AML interpreter's event log.

Syntax

```
!amli dl
```

DLL

Kdexts.dll

Additional Information

For information about related commands and their uses, see The AMLI Debugger.

Remarks

The event log chronicles the most recent 150 events that occurred in the interpreter.

Here is an example of the log display:

```
kd> !amli dl
RUN!: [c15a6618]QTh=00000000,QCt=00000000,QFg=00000000: Ctx=c18b4000,rc=0
KICK: [c15a6618]QTh=00000000,QCt=00000000,QFg=00000000: rc=0
SYNC: [c15a6618]QTh=00000000,QCt=00000000,QFg=00000002,LockPhase=0,Locked=0,IRQL=00: Obj=\_WAK
ASYN: [c15a6618]QTh=00000000,QCt=00000000,QFg=00000002,LockPhase=0,Locked=0,IRQL=00: Obj=\_WAK
REST: [c15a6618]QTh=00000000,QCt=00000000,QFg=00000002: Ctx=c18b4000,Obj=\_WAK
INSQ: [c15a6618]QTh=00000000,QCt=00000000,QFg=00000002: Ctx=c18b4000,Obj=\_WAK
EVAL: [c15a6618]QTh=00000000,QCt=00000000,QFg=00000002: Ctx=c18b4000,Obj=\_WAK
RUNC: [c15a6618]QTh=c15a6618,QCt=c18b4000,QFg=00000002: Ctx=c18b4000,Obj=\_WAK
```

!amli dns

12/5/2018 • 2 minutes to read • Edit Online

The **!amli dns** extension displays an ACPI namespace object.

Syntax

```
!amli dns [/s] [Name | Address]
```

Parameters

/s
Causes the entire namespace subtree under the specified object to be displayed recursively.

Name
Specifies the namespace path.

Address
Specifies the address of the namespace node.

DLL

Kdexts.dll

Additional Information

For information about related commands and their uses, see The AMLI Debugger.

Remarks

If neither *Name* nor *Address* is specified, the entire ACPI namespace tree is displayed recursively. The **/s** parameter is always assumed in this case, even if it is not specified.

This command is useful for determining what a particular namespace object is—whether it is a method, a field unit, a device, or another type of object.

Without the **/s** parameter, this extension is equivalent to the **!nsobj** extension. With the **/s** parameter, it is equivalent to the **!nstree** extension.

Here are some examples. The following command displays the namespace for the object **bios**:

```
AMLI(? for help)-> dns \bios

ACPI Name Space: \BIOS (80E5F378)
OpRegion(BIOS:RegionSpace=SystemMemory,Offset=0xfcb07500,Len=2816)
```

The following command displays the namespace for the object _BST, and the tree subordinate to it:

```
kd> !amli dns /s \_sb.pci0.isa.bat1._bst

ACPI Name Space: \_SB.PCI0.ISA.BAT1._BST (c29c2044)
Method(_BST:Flags=0x0,CodeBuff=c29c20a5,Len=103)
```

To display the namespace for the device BAT1, type:

```
kd> !amli dns /s \_sb.pci0.isa.bat1
```

To display the namespace of everything subordinate to the DOCK device, type:

```
kd> !amli dns /s \_sb.pci0.dock
```

To display the namespace subordinate to the _DCK method, type:

```
kd> !amli dns /s \_sb.pci0.dock._dck
```

To display the entire namespace, type:

```
kd> !amli dns
```

!amli do

12/5/2018 • 2 minutes to read • Edit Online

The **!amli do** extension displays an AML data object.

Syntax

```
!amli do Address
```

Parameters

Address
Specifies the address of the data object.

DLL

Kdexts.dll

Additional Information

For information about related commands and their uses, see The AMLI Debugger.

!amli ds

12/5/2018 • 2 minutes to read • Edit Online

The **!amli ds** extension displays an AML stack.

Syntax

```
!amli ds [/v] [Address]
```

Parameters

/v
Causes the display to be verbose. In Windows 2000, this option is available only if you are using the checked build of this extension (w2kchk\Acpikd.dll).

Address
Specifies the address of the context block whose stack is desired. If *Address* is omitted, the current context is used.

DLL

The !stacks extension displays information about the kernel stacks.

Additional Information

For information about related commands and their uses, see The AMLI Debugger.

!amli find

12/5/2018 • 2 minutes to read • Edit Online

The **!amli find** extension finds an ACPI namespace object.

Syntax

```
!amli find Name
```

Parameters

Name
Specifies the name of the namespace object (without the path).

DLL

Kdexts.dll

Additional Information

For information about related commands and their uses, see The AMLI Debugger.

Remarks

The **!amli find** command takes the name of the object and returns the full path and name. The *Name* parameter must be the final segment of the full path and name.

Here are some examples. The following command will find all declarations of the object _SRS:

```
kd> !amli find _srs
\_SB.LNKA._SRS
\_SB.LNKB._SRS
\_SB.LNKC._SRS
\_SB.LNKD._SRS
```

This is not simply a text search. The command **!amli find srs** does not display any hits, because the final segment of each of these declarations is "_SRS", not "SRS". The command **!amli find LNK** similarly does not return hits. The command **!amli find LNKB** would display the single node that terminates in "LNKB", not the four children of this node shown in the previous display:

```
kd> !amli find lnkb
\_SB.LNKB.
```

If you need to see the children of a node, use the **!amli dns** command with the **/s** parameter.

Here is another example, issued from the AMLI Debugger prompt. This shows all declarations of the object _BST in the namespace:

```
AMLI(? for help)-> find _bst
\_SB.PCI0.ISA.BAT1._BST
\_SB.PCI0.ISA.BAT2._BST
```

!amli lc

12/5/2018 • 2 minutes to read • Edit Online

The **!amli lc** extension lists all active ACPI contexts.

Syntax

```
!amli lc
```

DLL

Kdexts.dll

Additional Information

For information about related commands and their uses, see The AMLI Debugger.

Remarks

Each context corresponds to a method currently running in the AML interpreter.

Here is an example:

```
AMLI(? for help)-> lc
 Ctxt=80e3f000, ThID=00000000, Flgs=A--C-----, pbOp=00000000, Obj=\_SB.LNKA._STA
 Ctxt=80e41000, ThID=00000000, Flgs=A--C-----, pbOp=00000000, Obj=\_SB.LNKB._STA
 Ctxt=80e9a000, ThID=00000000, Flgs=A--C-----, pbOp=00000000, Obj=\_SB.LNKC._STA
 Ctxt=80ea8000, ThID=00000000, Flgs=A--C-----, pbOp=00000000, Obj=\_SB.LNKD._STA
*Ctxt=80e12000, ThID=80e6eda8, Flgs=---CR----, pbOp=80e5d5ac, Obj=\_SB.LNKA._STA
```

The **Obj** field gives the full path and name of the method as it appears in the ACPI tables.

The **Ctxt** field gives the address of the context block. The asterisk (*) *indicates the *current context*.* This is the context that was being executed by the interpreter when the break occurred.

The abbreviation **pbOp** indicates the instruction pointer (pointer to binary op codes).

There are nine flags that can be displayed in the **Flgs** section. If a flag is not set, a hyphen is displayed instead. The full list of flags is as follows:

FLAG	MEANING
A	Asynchronous evaluation
N	Nested evaluation
Q	In the ready queue
C	Needs a callback

FLAG	MEANING
R	Running
W	Ready
T	Time-out
D	Timer dispatch
P	Timer pending

!amli ln

12/5/2018 • 2 minutes to read • Edit Online

The **!amli ln** extension displays the specified method or the method containing a given address.

Syntax

```
!amli ln [ MethodName | CodeAddress ]
```

Parameters

MethodName
Specifies the full path of the method name. If *MethodName* specifies an object that is not actually a method, an error results.

CodeAddress
Specifies the address of the AML code that is contained in the desired method. If *CodeAddress* is prefixed with two percent signs (**%%**), it is interpreted as a physical address. Otherwise, it is interpreted as a virtual address.

DLL

Kdexts.dll

Additional Information

For information about related commands and their uses, see The AMLI Debugger.

Remarks

If neither *MethodName* nor *CodeAddress* is specified, the method associated with the current context is displayed.

The following command shows the method being currently run:

```
kd> !amli ln
c29accf5: \_WAK
```

The method _WAK is shown, with address 0xC29ACCF5.

!amli r

12/5/2018 • 2 minutes to read • Edit Online

The **!amli r** extension displays information about the current context or the specified context.

Syntax

```
!amli r [ContextAddress]
```

Parameters

ContextAddress
Specifies the address of the context block to be displayed. The address of a context block can be determined from the **Ctxt** field in the **!amli lc** display. If *ContextAddress* is prefixed with two percent signs (**%%**), it is interpreted as a physical address. Otherwise, it is interpreted as a virtual address. If this parameter is omitted, the current context is displayed.

DLL

Kdexts.dll

Additional Information

For information about related commands and their uses, see The AMLI Debugger.

Remarks

If the AMLI Debugger prompt appears suddenly, this is a useful command to use.

For example, the following command will display the current context of the interpreter:

```
AMLI(? for help)-> r

Context=c18b4000*, Queue=00000000, ResList=00000000
ThreadID=c15a6618, Flags=00000010
StackTop=c18b5eec, UsedStackSize=276 bytes, FreeStackSize=7636 bytes
LocalHeap=c18b40c0, CurrentHeap=c18b40c0, UsedHeapSize=88 bytes
Object=\_WAK, Scope=\_WAK, ObjectOwner=c18b4108, SyncLevel=0
AsyncCallBack=ff06b5d0, CallBackData=0, CallBackContext=c99efddc

MethodObject=\_WAK
80e0ff5c: Local0=Unknown()
80e0ff70: Local1=Unknown()
80e0ff84: Local2=Unknown()
80e0ff98: Local3=Unknown()
80e0ffac: Local4=Unknown()
80e0ffc0: Local5=Unknown()
80e0ffd4: Local6=Unknown()
80e0ffe8: Local7=Unknown()
80e0e040: RetObj=Unknown()

Next AML Pointer: ffffffff80e630df:[\_WAK+16]

ffffffff80e630df : If(S4BW
ffffffff80e630e5 : {
ffffffff80e630e5 : | Store(Zero, S4BW)
ffffffff80e630eb : }
```

!amli set

12/5/2018 • 2 minutes to read • Edit Online

The **!amli set** extension sets or displays the AMLI Debugger options.

```
!amli set Options
```

Parameters

Options
Specifies one or more options to be set. Separate multiple options with spaces. Possible values include:

spewon
Causes full debug output to be sent from the target computer. This option should be left on at all times for effective AML debugging. See the Remarks section for details.

spewoff
Suppresses debug output.

verboseon
Turns on verbose mode. This causes the AMLI Debugger to display the names of AML methods as they are evaluated.

verboseoff
Turns off verbose mode.

traceon
Activates ACPI tracing. This produces much more output than the **verboseon** option. This option is very useful for tracking SMI-related hard hangs.

traceoff
Deactivates ACPI tracing.

nesttraceon
Activates nest tracing. This option is only effective if the **traceon** option is also selected.

dbgbrkon
Enables breaking into the AMLI Debugger.

dbgbrkoff
Deactivates the dbgbrkon option.

nesttraceoff
Deactivates nest tracing.

lbrkon
Breaks into the AMLI Debugger when DDB loading is completed.

lbrkoff
Deactivates the **lbrkon** option.

errbrkon
Breaks into the AMLI Debugger whenever the interpreter has a problem evaluating AML code.

errbrkoff
Deactivates the **errbrkon** option.

logon
Enables event logging.

logoff
Disables event logging.

logmuton
Enables mutex event logging.

logmutoff
Disables mutex event logging.

DLL

Kdexts.dll

Additional Information

For information about related commands and their uses, see The AMLI Debugger.

Remarks

If no options are specified, the current status of all options is displayed.

By default, many messages are filtered out, you may need to turn this output on with **!amli set spewon**. Otherwise, numerous AMLI Debugger messages will be lost.

If the AML interpreter breaks into the AMLI Debugger, this output will be automatically turned on.

For more details on this output filtering, see **DbgPrintEx** and **KdPrintEx** in the Windows Driver Kit (WDK) documentation.

!amli u

12/5/2018 • 2 minutes to read • Edit Online

The **!amli u** extension unassembles AML code.

Syntax

```
!amli u [ MethodName | CodeAddress ]
```

Parameters

MethodName
Specifies the full path of the method name to be disassembled.

CodeAddress
Specifies the address of the AML code where disassembly will begin. If *CodeAddress* is prefixed with two percent signs (**%%**), it is interpreted as a physical address. Otherwise, it is interpreted as a virtual address.

DLL

Kdexts.dll

Additional Information

For information about related commands and their uses, see The AMLI Debugger.

Remarks

If neither *MethodName* nor *CodeAddress* is specified and you are issuing this command from an AMLI

The disassembly display will continue until the end of the method is reached.

Note The standard **u (Unassemble)** command will not give proper results with AML code.

Here are some examples. To disassemble the object at address 0x80E5D701, use the following command:

```
kd> !amli u 80e5d701

ffffffff80e5d701 : CreateWordField(CRES, 0x1, IRQW)
ffffffff80e5d70c : And(\_SB_.PCI0.LPC_.PIRA, 0xf, Local0)
ffffffff80e5d723 : Store(One, Local1)
ffffffff80e5d726 : ShiftLeft(Local1, Local0, IRQW)
ffffffff80e5d72d : Return(CRES)
```

The following command will disassemble the _DCK method:

```
kd> u \_sb.pci0.dock._dck
```

!facs

12/5/2018 • 2 minutes to read • Edit Online

The **!facs** extension displays a Firmware ACPI Control Structure (FACS).

Syntax

```
!facs Address
```

Parameters

Address
Specifies the address of the FACS.

DLL

Kdexts.dll

Additional Information

For more information, see ACPI Debugging.

!fadt

12/5/2018 • 2 minutes to read • Edit Online

The **!fadt** extension displays a Fixed ACPI Description Table (FADT).

Syntax

```
!fadt Address
```

Parameters

Address
Specifies the address of the FADT.

DLL

Kdexts.dll

Additional Information

For more information, see ACPI Debugging.

!mapic

12/5/2018 • 2 minutes to read • Edit Online

The **!mapic** extension displays an ACPI Multiple APIC table.

Syntax

```
!mapic Address
```

Parameters

Address
Specifies the address of the Multiple APIC Table.

DLL

Kdexts.dll

Additional Information

For more information, see ACPI Debugging.

!nsobj

12/5/2018 • 2 minutes to read • Edit Online

The **!nsobj** extension displays an ACPI namespace object.

Syntax

```
!nsobj [Address]
```

Parameters

Address
Specifies the address of the namespace object. If this is omitted, the root of the namespace tree is used.

DLL

Kdexts.dll

Additional Information

For more information, see ACPI Debugging.

Remarks

This extension is equivalent to **!amli dns**.

!nstree

12/5/2018 • 2 minutes to read • Edit Online

The **!nstree** extension displays an ACPI namespace object and its children in the namespace tree.

Syntax

```
!nstree [Address]
```

Parameters

Address
Specifies the address of the namespace object. This object and the entire namespace tree subordinate to it will be displayed. If *Address* is omitted, the entire namespace tree is displayed.

DLL

Kdexts.dll

Additional Information

For more information, see ACPI Debugging.

Remarks

This extension is equivalent to **!amli dns /s**.

!rsdt

12/5/2018 • 2 minutes to read • Edit Online

The **!rsdt** extension displays the ACPI Root System Description Table.

Syntax

```
!rsdt
```

DLL

Kdexts.dll

Additional Information

For more information, see ACPI Debugging.

Graphics Driver Extensions (Gdikdx.dll)

12/5/2018 • 2 minutes to read • Edit Online

Extension commands that are useful for debugging the Graphics Driver Interface (GDI) can be found in Gdikdx.dll.

The Windows 2000 version of this extension DLL appears in the w2kfre and w2kchk directories. There is no version of this DLL for Windows XP or later versions of Windows.

!gdikdx.verifier

12/5/2018 • 2 minutes to read • Edit Online

The **!gdikdx.verifier** extension displays the status of Driver Verifier during the verification of a graphics driver.

```
!gdikdx.verifier [-Flags]
```

Parameters

Flags
Specifies what information will be displayed in the output from this command. Any combination of the following (preceded by a hyphen) is allowed:

d
Causes the display to include statistics on **Memory Pool Tracking**. This includes the address, size, and tag of each pool.

h (or **?**)
Displays some brief Help text for this command in the Debugger Command window.

DLL

Windows 2000	Gdikdx.dll
Windows XP and later	Unavailable

Additional Information

For information about Driver Verifier, see the Windows Driver Kit (WDK) documentation.

Remarks

When verifying drivers that are not graphics drivers, the standard kernel-mode extension **!verifier** should be used instead of **!gdikdx.verifier**.

Regardless of which flags are selected, this extension will display the Driver Verifier options that are active. It will also display statistics on the frequency of random failure.

Kernel Streaming Extensions (Ks.dll)

12/5/2018 • 2 minutes to read • Edit Online

Extension commands that are useful for debugging kernel streaming drivers and AVStream drivers can be found in Ks.dll.

Most of the sample output in these reference pages was generated by debugging the filter-centric sample Avssamp.sys. This sample is included in the Windows Driver Kit.

If you wish to use the Ks.dll extensions with Windows 2000, you must copy Ks.dll from the kdexts directory to the w2kfre directory.

You need special symbols to use this extension. For more information, see the Debugging Tools for Windows Web site.

You can get additional information for many of the extension commands in this section simply by entering the command into the debugger with no arguments.

For more information, see Kernel Streaming Debugging.

!ks.help

12/5/2018 • 2 minutes to read • Edit Online

The **!ks.help** extension displays a help text showing all AVStream-specific Ks.dll extension commands.

```
!ks.help
```

DLL

Windows 2000	winxp\Ks.dll
Windows XP and later	Ks.dll

Additional Information

For more information, see Kernel Streaming Debugging.

!ks.kshelp

12/5/2018 • 2 minutes to read • Edit Online

The **!ks.kshelp** extension displays a help text showing original KS 1.0-specific Ks.dll extension commands.

```
!ks.kshelp
```

DLL

Windows 2000	winxp\Ks.dll
Windows XP and later	Ks.dll

Additional Information

For more information, see Kernel Streaming Debugging.

!ks.pchelp

12/5/2018 • 2 minutes to read • Edit Online

The **!ks.pchelp** extension displays a help text showing PortCls-specific Ks.dll extension commands.

```
!ks.pchelp
```

DLL

Windows 2000	winxp\Ks.dll
Windows XP and later	Ks.dll

Additional Information

For more information, see Kernel Streaming Debugging.

!ks.allstreams

12/5/2018 • 2 minutes to read • Edit Online

The **!ks.allstreams** extension walks the entire device tree and finds every kernel streaming device in the system.

```
!ks.allstreams [Flags] [Level]
```

Parameters

Flags
Optional. Specifies the kind of information to be displayed. *Flags* can be any combination of the following bits. The default value is 0x1:

Bit 0 (0x1)
Causes the display to include streams.

Bit 2 (0x4)
Causes the display to include proxy instances.

Bit 3 (0x8)
Causes the display to include queued IRPs.

Bit 4 (0x10)
Causes the display to include an unformatted display of all streams.

Bit 5 (0x20)
Causes the display to include an unformatted display of all stream formats.

Level
Optional. Specifies the level of detail to display on a 0-7 scale with progressively more information displayed for higher values. To display all available details, supply a value of 7.

DLL

Windows 2000	winxp\Ks.dll
Windows XP and later	Ks.dll

Additional Information

For more information, see Kernel Streaming Debugging.

Remarks

This command can take some time to execute (a minute is not unusual).

Here is an example of the **!ks.allstreams** display:

```
kd> !allstreams
6 Kernel Streaming FDOs found:
    Functional Device 82a17690 [\Driver\smwdm]
    Functional Device 8296eb08 [\Driver\wdmaud]
    Functional Device 82490388 [\Driver\sysaudio]
    Functional Device 82970cb8 [\Driver\MSPQM]
    Functional Device 824661b8 [\Driver\MSPCLOCK]
    Functional Device 8241c020 [\Driver\avssamp]
```

!ks.automation

12/5/2018 • 2 minutes to read • Edit Online

The **!ks.automation** extension displays any automation items associated with the given object.

```
!ks.automation Object
```

Parameters

Object
Specifies a pointer to the object for which to display automation items. (Automation items are properties, methods, and events.) *Object* must be one of the following types: PKSPIN, PKSFILTER, CKsPin*, CKsFilter*, PIRP. If *Object* is a pointer to an automation IRP, the command returns property information and handlers.

DLL

Windows 2000	winxp\Ks.dll
Windows XP and later	Ks.dll

Additional Information

For more information, see Kernel Streaming Debugging.

Remarks

You can use this command with a filter address obtained from **!ks.enumdevobj**.

Here is an example of the **!ks.automation** display. The argument is the address of a filter:

```
kd> !automation 829493c4
Filter 829493c4 has the following automation items:
    Property Items:
        Set KSPROPSETID_Pin
            Item ID = KSPROPERTY_PIN_CINSTANCES
                Get Handler = ks!CKsFilter::Property_Pin
                Set Handler = NULL
                MinProperty = 00000020
                MinData = 00000008
            Item ID = KSPROPERTY_PIN_CTYPES
                Get Handler = ks!CKsFilter::Property_Pin
                Set Handler = NULL
                MinProperty = 00000018
                MinData = 00000004
            Item ID = KSPROPERTY_PIN_DATAFLOW
                Get Handler = ks!CKsFilter::Property_Pin
                Set Handler = NULL
                MinProperty = 00000020
                MinData = 00000004
            Item ID = KSPROPERTY_PIN_DATARANGES
                Get Handler = ks!CKsFilter::Property_Pin
                Set Handler = NULL
                MinProperty = 00000020
                MinData = 00000000
```

```
            Item ID = KSPROPERTY_PIN_DATAINTERSECTION
                Get Handler = ks!CKsFilter::Property_Pin
                Set Handler = NULL
                MinProperty = 00000028
                MinData = 00000000
            Item ID = KSPROPERTY_PIN_INTERFACES
                Get Handler = ks!CKsFilter::Property_Pin
                Set Handler = NULL
                MinProperty = 00000020
                MinData = 00000000
            Item ID = KSPROPERTY_PIN_MEDIUMS
                Get Handler = ks!CKsFilter::Property_Pin
                Set Handler = NULL
                MinProperty = 00000020
                MinData = 00000000
            Item ID = KSPROPERTY_PIN_COMMUNICATION
                Get Handler = ks!CKsFilter::Property_Pin
                Set Handler = NULL
                MinProperty = 00000020
                MinData = 00000004
            Item ID = KSPROPERTY_PIN_NECESSARYINSTANCES
                Get Handler = ks!CKsFilter::Property_Pin
                Set Handler = NULL
                MinProperty = 00000020
                MinData = 00000004
            Item ID = KSPROPERTY_PIN_CATEGORY
                Get Handler = ks!CKsFilter::Property_Pin
                Set Handler = NULL
                MinProperty = 00000020
                MinData = 00000010
            Item ID = KSPROPERTY_PIN_NAME
                Get Handler = ks!CKsFilter::Property_Pin
                Set Handler = NULL
                MinProperty = 00000020
                MinData = 00000000
        Set KSPROPSETID_Topology
            Item ID = KSPROPERTY_TOPOLOGY_CATEGORIES
                Get Handler = ks!CKsFilter::Property_Topology
                Set Handler = NULL
                MinProperty = 00000018
                MinData = 00000000
            Item ID = KSPROPERTY_TOPOLOGY_NODES
                Get Handler = ks!CKsFilter::Property_Topology
                Set Handler = NULL
                MinProperty = 00000018
                MinData = 00000000
            Item ID = KSPROPERTY_TOPOLOGY_CONNECTIONS
                Get Handler = ks!CKsFilter::Property_Topology
                Set Handler = NULL
                MinProperty = 00000018
                MinData = 00000000
            Item ID = KSPROPERTY_TOPOLOGY_NAME
                Get Handler = ks!CKsFilter::Property_Topology
                Set Handler = NULL
                MinProperty = 00000020
                MinData = 00000000
        Set KSPROPSETID_General
            Item ID = KSPROPERTY_GENERAL_COMPONENTID
                Get Handler = ks!CKsFilter::Property_General_ComponentId
                Set Handler = NULL
                MinProperty = 00000018
                MinData = 00000048
        Set [ks!KSPROPSETID_Frame]  a60d8368-5324-4893-b020-c431a50bcbe3
            Item ID = 0
                Get Handler = ks!CKsFilter::Property_Frame_Holding
                Set Handler = ks!CKsFilter::Property_Frame_Holding
                MinProperty = 00000018
                MinData = 00000004
    Method Items:
```

```
        NO SETS FOUND!
    Event Items:
        NO SETS FOUND!
```

!ks.devhdr

12/5/2018 • 2 minutes to read • Edit Online

The **!ks.devhdr** extension displays the kernel streaming device header associated with the given WDM object.

```
!ks.devhdr DeviceObject
```

Parameters

DeviceObject
This parameter specifies a pointer to a WDM device object. If *DeviceObject* is not valid, the command returns an error.

DLL

Windows 2000	winxp\Ks.dll
Windows XP and later	Ks.dll

Additional Information

For more information, see Kernel Streaming Debugging.

Remarks

The output from **!ks.allstreams** can be used as the input for **!ks.devhdr**.

Here is an example of the **!ks.devhdr** display:

```
kd> !devhdr 827aedf0 7
Device Header 824ca1e0
    Child Create Handler List:
        Create Item eb3a7284
            CreateFunction = sysaudio!CFilterInstance::FilterDispatchCreate+0x00
            ObjectClass = NULL
            Flags = 0
```

!ks.dhdr

12/5/2018 • 2 minutes to read • Edit Online

The **!ks.dhdr** extension command is obsolete; use **!ks.dump** instead.

!ks.dump

3/5/2019 • 2 minutes to read • Edit Online

The **!ks.dump** extension displays the specified object.

```
!ks.dump Object [Level] [Flags]
```

Parameters

Object
Specifies a pointer to an AVStream structure, an AVStream class object, or a PortCls object. Can also specify a pointer to an IRP or a file object.

Level
Optional. Specifies the level of detail to display on a 0-7 scale with progressively more information displayed for higher values. To display all available details, supply a value of 7. You can see more information about levels by issuing a **!ks.dump** command with no arguments.

Flags
Optional. Specifies the kind of information to be displayed. *Flags* can be any combination of the following bits.

Bit 0 (0x1)
Display all queued IRPs.

Bit 1 (0x2)
Display all pending IRPs.

Bit 2 (0x4)
Analyze a stalled graph for suspects.

Bit 3 (0x8)
Show all pin states.

DLL

Windows 2000	winxp\Ks.dll
Windows XP and later	Ks.dll

Additional Information

For more information, see Kernel Streaming Debugging.

Remarks

The **!ks.dump** command recognizes most AVStream objects, including pins, filters, factories, devices, pipes, and stream pointers. This command also recognizes some stream class structures, including stream objects, filter instances, device extensions, and SRBs.

Following is an example of the **!ks.dump** display for a filter:

```
kd> !dump 829493c4
Filter object 829493c4 [CKsFilter = 82949350]
    Descriptor     f7a233c8:
    Context        829dce28
```

Following is an example of the **!ks.dump** display for a pin:

```
kd> !dump 8160DDE0 7
Pin object 8160DDE0 [CKsPin = 8160DD50]
    DeviceState    KSSTATE_RUN
    ClientState    KSSTATE_RUN
    ResetState     KSRESET_END
    CKsPin object 8160DD50 [KSPIN = 8160DDE0]
        State                    KSSTATE_RUN
        Processing Mutex         8160DFD0 is not held
        And Gate &               8160DF88
        And Gate Count           1
```

Some important parts of this display are included in the following table.

Parameter	**Meaning**
DeviceState	The state that the pin was requested to enter. If different from ClientState, this is the state that the minidriver will transition to next.
ClientState	The state that the minidriver is actually in. This reflects the state of the pipe.
ResetState	Indicates whether or not the object is in the middle of a flush. KSRESET_BEGIN indicates a flush. KSRESET_END indicates no flush.
State	The internal state of the pin's transport to non-AVStream filters.

Following is an example of the **!ks.dump** display for a stream class driver:

```
kd> !dump 81a0a170 7
Device Extension 81a0a228:
    Device Object          81a0a170 [\Driver\TESTCAP]
    Next Device Object     81bd56d8 [\Driver\PnpManager]
    Physical Device Object 81bd56d8 [\Driver\PnpManager]
    REGISTRY FLAGS:
        Page out driver when closed
        No suspend if running
    MINIDRIVER Data:
        Device Extension       81a0a44c
        Interrupt Routine      00000000
        Synchronize Routine    STREAM!StreamClassSynchronizeExecution
        Receive Device SRB     testcap!AdapterReceivePacket
        Cancel Packet          testcap!AdapterCancelPacket
        Timeout Packet         testcap!AdapterTimeoutPacket
        Size (d / r / s / f)   1a0(416), 14(20), 978(2424), 0(0)
        Sync Mode              Driver Synchronizes
    Filter Type 0:
        Symbolic Links:
            Information Paged Out
        Instances:
            816b7bd8
```

Note that the sizes are listed both in hexadecimal numbers, and then, parenthetically in the decimal equivalent. The Size abbreviations in this display are listed in the following table.

Size	**Explanation**
d	Device
r	Request
s	Stream
f	Filter. If the filter size is 0, the filter is single instance. If it is greater than 0, it is multi-instance.

!ks.dumpbag

12/5/2018 • 2 minutes to read • Edit Online

The **!ks.dumpbag** extension displays the contents of the object bag for the specified object.

```
!ks.dumpbag Object [Level]
```

Parameters

Object
Specifies a pointer to a valid client viewable object structure, or to the private class object.

Level
Optional. Specifies the level of detail to display on a 0-7 scale with progressively more information displayed for higher values. To display all available details, supply a value of 7.

DLL

Windows 2000	winxp\Ks.dll
Windows XP and later	Ks.dll

Additional Information

For more information, see Kernel Streaming Debugging.

Remarks

Here is an example of the **!ks.dumpbag** display for a filter:

```
kd> !dumpbag 829493c4
Filter 829493c4 [CKsFilter = 82949350]:
    Object Bag 829493d0:
        Object Bag Item 829dce28:
            Reference Count        : 1
            Item Cleanup Handler   : f7a21730
```

!ks.dumpcircuit

12/5/2018 • 2 minutes to read • Edit Online

The **!ks.dumpcircuit** extension lists details of the transport circuit associated with the given object.

```
!ks.dumpcircuitextension Object [Level]
```

Parameters

Object
Specifies a pointer to the object for which to display the transport circuit. For AVStream, *Object* must be one of the following types: CKsPin*, CKsQueue*, CKsRequestor*, CKsSplitter*, CKsSplitterBranch*.

For PortCls, object must be one of the following types: CPortPin*, CKsShellRequestor*, or CIrpStream*.

Level
Optional. Specifies the level of detail to display on a 0-7 scale with progressively more information displayed for higher values. To display all available details, supply a value of 7.

DLL

Windows 2000	winxp\Ks.dll
Windows XP and later	Ks.dll

Additional Information

For more information, see Kernel Streaming Debugging.

Remarks

Note that **!ks.dumpcircuit** starts walking the circuit at the specified object, which does not always correspond to the data source.

You can first use **!ks.graph** with a filter address to list pin addresses, and then use these addresses with **!ks.dumpcircuit**.

Here is an example of the **!ks.dumpcircuit** display:

```
kd> !dumpcircuit 8293f4f0
Pin8293f4f0 0 (snk, out)
Queue82990e20 r/w/c=2489/2/0
```

!ks.dumplog

12/5/2018 • 2 minutes to read • Edit Online

The **!ks.dumplog** extension displays the internal kernel streaming debug log.

```
!ks.dumplog [Entries]
```

Parameters

Entries
Optional. Specifies the number of log entries to display. If *Entries* is zero or omitted, the entire log is displayed.

DLL

Windows 2000	winxp\Ks.dll
Windows XP and later	Ks.dll

Additional Information

For more information, see Kernel Streaming Debugging.

Remarks

You can stop the log display by pressing **CTRL+C**.

This extension requires that the target computer be running a checked (debug) version of Ks.sys.

!ks.dumpqueue

12/5/2018 • 2 minutes to read • Edit Online

The **!ks.dumpqueue** extension displays information about the queues associated with a given AVStream object, or the stream associated with a port class object.

```
!ks.dumpqueue Object [Level]
```

Parameters

Object
Specifies a pointer to the object for which to display the queue. *Object* must be of type PKSPIN, PKSFILTER, CKsPin*, CKsFilter*, CKsQueue*, CPortPin*, or CPortFilter*.

Level
Optional. Specifies the level of detail to display on a 0-7 scale with progressively more information displayed for higher values. To display all available details, supply a value of 7.

DLL

Windows 2000	winxp\Ks.dll
Windows XP and later	Ks.dll

Additional Information

For more information, see Kernel Streaming Debugging.

Remarks

Object must be a filter or a pin. For a pin, a single queue is displayed. For a filter, multiple queues are displayed.

This command can take some time to execute.

Here is an example of the **!ks.dumpqueue** display:

```
kd> !dumpqueue 829493c4
Filter 829493c4: Output Queue 82990e20:
 Queue 82990e20:
        Frames Received  : 1889
        Frames Waiting   : 3
        Frames Cancelled : 0
        And Gate 82949464 : count = 1, next = 00000000
    Frame Gate NULL
        Frame Header 82aaef78:
 NextFrameHeaderInIrp = 00000000
            OriginalIrp = 82169e48
            Mdl = 8292e358
            Irp = 82169e48
 StreamHeader = 8298dea0
            FrameBuffer = edba3000
            StreamHeaderSize = 00000000
            FrameBufferSize = 00025800
            Context = 00000000
            Refcount = 1
```

!ks.enumdevobj

12/5/2018 • 2 minutes to read • Edit Online

The **!ks.enumdevobj** extension displays the KSDEVICE associated with a given WDM device object, and lists the filter types and filters currently instantiated on this device.

```
!ks.enumdevobj DeviceObject
```

Parameters

DeviceObject
Specifies a pointer to a WDM device object.

DLL

Windows 2000	winxp\Ks.dll
Windows XP and later	Ks.dll

Additional Information

For more information, see Kernel Streaming Debugging.

Remarks

The output from **!ks.allstreams** can be used as the input for **!ks.enumdevobj**.

Here is an example of the **!ks.enumdevobj** display:

```
kd> !enumdevobj 8241c020
WDM device object 8241c020:
    Corresponding KSDEVICE        823b8430
    Factory 829782dc [Descriptor f7a233c8] instances:
        829493c4
```

!ks.enumdrvobj

12/5/2018 • 2 minutes to read • Edit Online

The **!ks.enumdrvobj** extension displays all KSDEVICE structures associated with a given WDM driver object, and lists the filter types and filters currently instantiated on these devices.

```
!ks.enumdrvobj DriverObject
```

Parameters

DriverObject
Specifies a pointer to a WDM driver object.

DLL

Windows 2000	winxp\Ks.dll
Windows XP and later	Ks.dll

Additional Information

For more information, see Kernel Streaming Debugging.

Remarks

Since **!ks.enumdrvobj** enumerates every device chained off a WDM driver object, it is equivalent to invoking **!ks.enumdevobj** on every device chained off a given driver.

!ks.eval

12/5/2018 • 2 minutes to read • Edit Online

The **!ks.eval** extension evaluates an expression using an extension-specific expression evaluator.

```
!ks.eval Expression
```

Parameters

Expression
Specifies the expression to evaluate. *Expression* can include any MASM operators, symbols, or numerical syntax, as well as the extension-specific operators described below. For more information about MASM expressions, see MASM Numbers and Operators.

DLL

Windows 2000	winxp\Ks.dll
Windows XP and later	Ks.dll

Additional Information

For more information, see Kernel Streaming Debugging.

Remarks

The extension module includes two extension-specific operators which can be used in address parameters to extension commands:

fdo(*x* **)**
Returns the functional device object associated with the object at address **x**.

driver(*x* **)**
Returns the driver object associated with **fdo(***x***)**.

You can use the **!ks.eval** command to parse expressions that contain these extension-specific operators as well as MASM Numbers and Operators.

Note that all operators supported by **!ks.eval** are also supported by all other extension commands in the Ks.dll extension module.

Here is an example of the **!ks.eval** extension being used with the address of a filter. Note the presence of the 0x8241C020 address in the **!ks.allstreams** output:

```
kd> !eval fdo(829493c4)
Resulting Evaluation: 8241c020

kd> !allstreams
6 Kernel Streaming FDOs found:
    Functional Device 82a17690 [\Driver\smwdm]
    Functional Device 8296eb08 [\Driver\wdmaud]
    Functional Device 82490388 [\Driver\sysaudio]
    Functional Device 82970cb8 [\Driver\MSPQM]
    Functional Device 824661b8 [\Driver\MSPCLOCK]
    Functional Device 8241c020 [\Driver\avssamp]
```

!ks.findlive

12/5/2018 • 2 minutes to read • Edit Online

The **!ks.findlive** extension searches the internal Ks.sys debug log to attempt to find live objects of a specified type.

```
!ks.findlive Type [Entries] [Level]
```

Parameters

Type
Specifies the type of object for which to search. Enter one of the following as a literal value: **Queue**, **Requestor**, **Pin**, **Filter**, or **Irp**.

Entries
If this parameter is zero or omitted, the entire log is searched.

Level
Optional. Specifies the level of detail to display on a 0-7 scale with progressively more information displayed for higher values. To display all available details, supply a value of 7.

DLL

Windows 2000	winxp\Ks.dll
Windows XP and later	Ks.dll

Additional Information

For more information, see Kernel Streaming Debugging.

Remarks

The **!ks.findlive** command may not find all possible specified live objects.

This extension requires that the target computer be running a checked (debug) version of Ks.sys.

!ks.forcedump

12/5/2018 • 2 minutes to read • Edit Online

The **!ks.forcedump** command displays information about memory contents at a caller-supplied address.

```
!ks.forcedump Object Type [Level]
```

Parameters

Object
Specifies a pointer to the object for which to display information.

Type
Specifies the type of object.

For AVStream/KS objects, *Type* must be one of the following values: CKsQueue, CKsDevice, CKsFilterFactory, CKsFilter, CKsPin, CKsRequestor, CKsSplitter, CKsSplitterBranch, CKsPipeSection, KSPIN, KSFILTER, KSFILTERFACTORY, KSDEVICE, KSSTREAM_POINTER, KSPFRAME_HEADER, KSIOBJECT_HEADER, KSPDO_EXTENSION, KSIDEVICE_HEADER, KSSTREAM_HEADER, KSPIN_DESCRIPTOR_EX, CKsProxy, CKsInputPin, CKsOutputPin, CasyncItemHandler.

For Port Class objects, *Type* must be one of the following values: DEVICE_CONTEXT, CPortWaveCyclic, CPortPinWaveCyclic, CPortTopology, CPortDMus, CIrpStream, CKsShellRequestor, CPortFilterWaveCyclic, CDmaChannel, CPortWavePci, CportPinWavePci.

Level
Optional. Specifies the level of detail to display on a 0-7 scale with progressively more information displayed for higher values. To display all available details, supply a value of 7.

DLL

Windows 2000	winxp\Ks.dll
Windows XP and later	Ks.dll

Additional Information

For more information, see Kernel Streaming Debugging.

Remarks

Normally, you can use **!ks.dump** to display data structures.

However, if symbols are loaded incorrectly or too much information is paged out, the type identification logic in the **!ks.dump** command may fail to identify the type of structure at a given address.

If this happens, try using the **!ks.forcedump** command. This command works just like **!ks.dump** except that the user specifies the type of the object.

Note The **!ks.forcedump** command does not verify that *Type* is the correct type of structure found at the address provided in *Object*. The command assumes that this is the type of structure found at *Object* and displays data

accordingly.

A listing of all supported objects can be retrieved by issuing a **!ks.forcedump** command with no arguments.

Here are two examples of the output from **!ks.forcedump**, using the address of a filter for the *Object* argument but with different levels of detail:

```
kd> !forcedump 829493c4 KSFILTER
WARNING: I am dumping 829493c4 as a KSFILTER.
         No checking has been performed to ensure that it is this type!!!

Filter object 829493c4 [CKsFilter = 82949350]
    Descriptor     f7a233c8:
    Context        829dce28

kd> !forcedump 829493c4 KSFILTER 7
WARNING: I am dumping 829493c4 as a KSFILTER.
         No checking has been performed to ensure that it is this type!!!

Filter object 829493c4 [CKsFilter = 82949350]
    Descriptor     f7a233c8:
    Filter Category GUIDs:
        Video
        Capture
    Context        829dce28
    INTERNAL INFORMATION:
        Public Parent Factory   829782dc
        Aggregated Unknown      00000000
        Device Interface        823b83c8
        Control Mutex           829493f8 is not held
    Object Event List:
        None
    CKsFilter object 82949350 [KSFILTER = 829493c4]
        Processing Mutex         82949484 is not held
        Gate &                   82949464
        Gate.Count               1
        Pin Factories:
            Pin ID 0 [Video/General Capture Out]:
                Child Count        1
                Bound Child Count  1
                Necessary Count    0
                Specific Instances:
                    8293f580
```

!ks.graph

12/5/2018 • 2 minutes to read • Edit Online

The **!ks.graph** extension command displays a textual description of the kernel mode graph in topologically sorted order.

```
!ks.graph Object [Level] [Flags]
```

Parameters

Object
Specifies a pointer to the object to use as a starting point for the graph. Must be a pointer to one of the following: file object, IRP, pin, or filter.

Level
Optional. Specifies the level of detail to display on a 0-7 scale with progressively more information displayed for higher values. To display all available details, supply a value of 7. The levels for **!ks.graph** are the same as those for **!ks.dump**.

Flags
Optional. Specifies the kind of information to be displayed. *Flags* can be any combination of the following bits.

Bit 0 (0x1)
Display a list of IRPs queued to each pin instance in the graph.

Bit 1 (0x2)
Display a list of IRPs that are pending from each pin instance in the graph. Only IRPs that the pin knows it is waiting for are displayed.

Bit 4 (0x10)
Analyze a stalled graph for suspect filters.

DLL

Windows 2000	winxp\Ks.dll
Windows XP and later	Ks.dll

Additional Information

For more information, see Kernel Streaming Debugging.

Remarks

This command may take a bit of time to process.

Issue a **!ks.graph** command with no arguments for help.

Here is an example of the **!ks.graph** display, with the address of a filter object:

```
kd> !graph 829493c4
Attempting a graph build on 829493c4...  Please be patient...

Graph With Starting Point 829493c4:

"avssamp" Filter 82949350, Child Factories 1
    Output Factory 0 [Video/General Capture]:
        Pin 8293f4f0 (File 82503498) Irps(q/p) = 2, 0
```

!ks.libexts

12/5/2018 • 2 minutes to read • Edit Online

The **!ks.libexts** extension provides access to Microsoft-supplied library extensions that are statically linked to the extension module.

```
!ks.libexts [Command] [Libext]
```

Parameters

Command
Optional. Specifies one of the following values. If this argument is omitted, **!ks.libexts** returns help information.

disableall
Disable all library extensions. When this is used, omit the *Libext* parameter.

disable
Disable a specific library extension by name. When this is used, specify the name in the *Libext* parameter.

enableall
Enable all library extensions. Only loaded components with correct symbols are enabled. When this is used, omit the *Libext* parameter.

enable
Enable a specific library extension by name. When this is used, specify the name in the *Libext* parameter. Only loaded components with correct symbols can be enabled.

details
Show details about all currently linked library extensions. When this is used, omit the *Libext* parameter.

Libext
Specifies the name of a library extension. Required only for *Command* values of **enable** or **disable**.

DLL

Windows 2000	winxp\Ks.dll
Windows XP and later	Ks.dll

Additional Information

For more information, see Kernel Streaming Debugging.

Remarks

The extension module contains an extensibility framework that allows separate components to be built and linked into Ks.dll. These extra components are called library extensions.

The **!ks.libexts** command allows viewing of statistics about those library extensions as well as control over them. For details, issue **!ks.libexts** with no arguments.

!ks.objhdr

12/5/2018 • 2 minutes to read • Edit Online

The **!ks.objhdr** extension displays the kernel streaming object header associated with the specified file object.

```
!ks.objhdr FileObject [Level] [Flags]
```

Parameters

FileObject
This parameter specifies a pointer to a WDM file object. If *FileObject* is not valid, the command returns an error.

Level
Optional. Values are the same as those for **!ks.dump**.

Flags
Optional. Values are the same as those for **!ks.dump**.

DLL

Windows 2000	winxp\Ks.dll
Windows XP and later	Ks.dll

Additional Information

For more information, see Kernel Streaming Debugging.

Remarks

Levels and flags for **!ks.objhdr** are identical to those described in **!ks.dump**.

The output from **!ks.allstreams** and **!ks.enumdevobj** can be used as the input for **!ks.objhdr**. To do this with the *avssamp* sample, for instance, issue the following commands:

```
kd> !ks.allstreams
6 Kernel Streaming FDOs found:
    Functional Device 8299be18 [\Driver\smwdm]
    Functional Device 827c86d8 [\Driver\wdmaud]
    Functional Device 827c0f08 [\Driver\sysaudio]
    Functional Device 82424590 [\Driver\avssamp]
    Functional Device 82423720 [\Driver\MSPQM]
    Functional Device 82b91a88 [\Driver\MSPCLOCK]
kd> !ks.enumdevobj 82424590
WDM device object 82424590:
    Corresponding KSDEVICE        82427540
    Factory 8285baa4 [Descriptor f7a333c8] instances:
        82837a34
kd> !ks.objhdr 82837a34 7
```

The results of this command might be lengthy. Issue a Ctrl-BREAK (WinDbg) or Ctrl-C (NTSD, CDB, KD) to stop the output.

Here's a separate example:

```
kd> !ks.objhdr 81D828B8 7
Adjusting file object 81D828B8 to object header 81BC1008

Object Header 81BC1008
    Associated Create Items:
        Create Item F9F77E98
            CreateFunction = ks!CKsFilter::DispatchCreatePin+0x00
            ObjectClass = {146F1A80-4791-11D0-A5D6-28DB04C10000}
            Flags = 0
    Child Create Handler List:
        Create Item F9F85AA0
            CreateFunction = ks!CKsPin::DispatchCreateAllocator+0x00
            ObjectClass = {642F5D00-4791-11D0-A5D6-28DB04C10000}
            Flags = 0
        Create Item F9F85AB8
            CreateFunction = ks!CKsPin::DispatchCreateClock+0x00
            ObjectClass = {53172480-4791-11D0-A5D6-28DB04C10000}
            Flags = 0
        Create Item F9F85AD0
            CreateFunction = ks!CKsPin::DispatchCreateNode+0x00
            ObjectClass = {0621061A-EE75-11D0-B915-00A0C9223196}
            Flags = 0
    DispatchTable:
        Dispatch Table F9F85AE8
            DeviceIoControl = ks!CKsPin::DispatchDeviceIoControl+0x00
            Read = ks!KsDispatchInvalidDeviceRequest+0x00
            Write = ks!KsDispatchInvalidDeviceRequest+0x00
            Flush = ks!KsDispatchInvalidDeviceRequest+0x00
            Close = ks!CKsPin::DispatchClose+0x00
            QuerySecurity = ks!KsDispatchQuerySecurity+0x00
            SetSecurity = ks!KsDispatchSetSecurity+0x00
            FastDeviceIoControl = ks!KsDispatchFastIoDeviceControlFailure+0x00
            FastRead = ks!KsDispatchFastReadFailure+0x00
            FastWrite = ks!KsDispatchFastReadFailure+0x00
    TargetState: KSTARGET_STATE_ENABLED
    TargetDevice: 00000000
    BaseDevice  : 81BBDF10
    Stack Depth : 1
    Corresponding AVStream object = 81A971B0
```

!ks.ohdr

12/5/2018 • 2 minutes to read • Edit Online

The **!ks.ohdr** extension displays details of a kernel streaming object header.

```
!ks.ohdr Object [Level] [Flags]
```

Parameters

Object
This parameter specifies a pointer to a KS object header. If *Object* is not valid, the command returns an error.

Level
Optional. Values are the same as those for **!ks.dump**.

Flags
Optional. Values are the same as those for **!ks.dump**.

DLL

Windows 2000	winxp\Ks.dll
Windows XP and later	Ks.dll

Additional Information

For more information, see Kernel Streaming Debugging.

Remarks

The **!ks.ohdr** command works similarly to **!ks.objhdr** in that it displays details of a KS object header. The difference is that the caller provides the direct address of the KS object header, instead of the address of the associated file object.

Levels and flags for **!ks.ohdr** are identical to those described in **!ks.dump**.

If the data you are querying is not paged out, consider using **!ks.dump** instead of **!ks.ohdr**.

!ks.pciaudio

12/5/2018 • 2 minutes to read • Edit Online

The **!ks.pciaudio** extension displays a list of FDOs currently attached to PortCls.

```
!ks.pciaudio [Options] [Level]
```

Parameters

Options
Optional. Specifies the kind of information to be displayed. *Options* can be any combination of the following bits.

Bit 0 (0x1)
Display a list of running streams.

Bit 1 (0x2)
Display a list all streams.

Bit 3 (0x4)
Output displayed streams. *Level* has meaning only when this bit is set.

Level
Optional, and applicable only if Bit 3 is set in *Options*. Levels are the same as those for **!ks.dump**.

DLL

Windows 2000	winxp\Ks.dll
Windows XP and later	Ks.dll

Additional Information

For more information, see Kernel Streaming Debugging.

Remarks

Here is an example of the output from **!ks.pciaudio**:

```
kd> !ks.pciaudio
1 Audio FDOs found:
 Functional Device 8299be18 [\Driver\smwdm]
```

!ks.pciks

12/5/2018 • 2 minutes to read • Edit Online

The **!ks.pciks** extension lists functional devices for kernel streaming devices that are attached to the PCI bus. Optionally, it can display information about active streams on those functional devices.

```
!ks.pciks [Flags] [Level]
```

Parameters

Flags
Optional. Specifies the kind of information to be displayed. *Flags* can be any combination of the following bits.

Bit 0 (0x1)
List all currently running streams.

Bit 1 (0x2)
Recurse graphs to find non-PCI devices.

Bit 2 (0x4)
Display a list of proxy instances.

Bit 3 (0x8)
Display currently queued Irps.

Bit 4 (0x10)
Display information about all streams.

Bit 5 (0x20)
Display active stream formats.

Level
Optional, and applicable only to flag combinations that cause data to be displayed. Levels are the same as those for **!ks.dump**.

DLL

Windows 2000	winxp\Ks.dll
Windows XP and later	Ks.dll

Additional Information

For more information, see Kernel Streaming Debugging.

Remarks

This command may take time to execute, especially if the ACPI filter driver is loaded, or if Driver Verifier is enabled and driver names are paged out.

Here is an example of the **!ks.pciks** display:

```
kd> !pciks
1 Kernel Streaming FDOs found:
    Functional Device 82a17690 [\Driver\smwdm]
```

!ks.shdr

12/5/2018 • 2 minutes to read • Edit Online

The **!ks.shdr** extension command is obsolete; use **!ks.dump** instead.

!ks.topology

12/5/2018 • 2 minutes to read • Edit Online

The **!ks.topology** extension displays a sorted graph of the internal topology of the filter closest to *Object*.

```
!ks.topology Object [Level] [Flags]
```

Parameters

Object
Specifies a pointer to the object to use as a base for the graph. Can be a pointer to a file object, IRP, pin, filter, or other KS object.

Level
Optional. Specifies the level of detail to display on a 0-7 scale with progressively more information displayed for higher values. To display all available details, supply a value of 7.

Flags
Not currently available.

DLL

Windows 2000	winxp\Ks.dll
Windows XP and later	Ks.dll

Additional Information

For more information, see Kernel Streaming Debugging.

Remarks

For help, issue a **!ks.topology** command with no arguments.

Note that this command may take a few moments to execute.

SCSI Miniport Extensions (Scsikd.dll and Minipkd.dll)

12/5/2018 • 2 minutes to read • Edit Online

Extension commands that are useful for debugging SCSI miniport drivers can be found in Scsikd.dll and Minipkd.dll.

You can use the Scsikd.dll extension commands with any version of Windows. However, you can only use the Minipkd.dll extension commands with Windows XP and later versions of Windows. Commands in Minipkd.dll are only applicable to SCSIport-based miniports.

For more information, see SCSI Miniport Debugging.

!scsikd.help

12/5/2018 • 2 minutes to read • Edit Online

The **!scsikd.help** extension displays help text for Scsikd.dll extension commands.

```
!scsikd.help
```

DLL

Windows 2000	Scsikd.dll
Windows XP and later	Scsikd.dll

Additional Information

For more information, see SCSI Miniport Debugging.

!scsikd.classext

12/5/2018 • 2 minutes to read • Edit Online

The **!scsikd.classext** extension displays the specified class Plug and Play (PnP) device.

```
!scsikd.classext [Device [Level]]
```

Parameters

Device
Specifies the device object or device extension of a class PnP device. If *Device* is omitted, a list of all class PnP extensions is displayed.

Level
Specifies the amount of detail to display. This parameter can take 0, 1, or 2 as values, with 2 giving the most detail. The default is 0.

DLL

Windows 2000	Scsikd.dll
Windows XP and later	Scsikd.dll

Additional Information

For more information, see SCSI Miniport Debugging.

Remarks

Here is an example of the **!scsikd.classext** display:

```
0: kd> !scsikd.classext
  ' !scsikd.classext 8633e3f0 '   (             ) "IBM     " / "DDYS-T09170M    " / "S93E" / "
XBY45906"
  ' !scsikd.classext 86347b48 '   (paging device) "IBM     " / "DDYS-T09170M    " / "S80D" / "
VDA60491"
  ' !scsikd.classext 86347360 '   (             ) "UNISYS  " / "003451ST34573WC " / "5786" /
"HN0220750000181300L6"
  ' !scsikd.classext 861d1898 '   (             ) "" / "MATSHITA CD-ROM CR-177" / "7T03" / ""

 usage: !classext <class fdo> <level [0-2]>
```

!scsikd.scsiext

12/5/2018 • 2 minutes to read • Edit Online

The **!scsikd.scsiext** extension displays detailed information about the specified SCSI port extension.

```
!scsikd.scsiext Device
```

Parameters

Device
Specifies the device object or device extension of a SCSI port extension.

DLL

Windows 2000	Scsikd.dll
Windows XP and later	Scsikd.dll

Additional Information

For more information, see SCSI Miniport Debugging.

Remarks

Here is an example of the **!scsikd.scsiext** display, where the SCSI port extension has been specified by a functional device object (FDO); this can be obtained from the **DO** field or **DevExt** field in the **!minipkd.adapters** display:

```
kd> !scsikd.scsiext 816f9a40
Scsiport functional device extension at address 816f9af8
Common Extension:
  Initialized
  DO 0x816f9a40  LowerObject 0x816e8030  SRB Flags 00000000
  Current Power (D0,S0)  Desired Power D-1 Idle 00000000
  Current PnP state 0x0    Previous state 0xff
  DispatchTable f9aee200   UsePathCounts (P0, H0, C0)
Adapter Extension:
  Port 2     IsPnp VirtualSlot HasInterrupt
  LowerPdo 0x816e8030   HwDevExt 0x8170a004   Active Requests 0xffffffff
  MaxBus 0x01   MaxTarget 0x10   MaxLun 0x08
  Port Flags (0x00001000): PD_DISCONNECT_RUNNING
  NonCacheExt 0x81702000  IoBase 0x00002000   Int 0x1a
  RealBus# 0x0  RealSlot# 0x2
  Timeout 0xffffffff   DpcFlags 0x00000000   Sequence 0x00000003
  Srb Ext Header 0x817061a0   No. Requests 0x00000012
  QueueTag BitMap 0x00000000   Hint 0x00000000
  MaxQueueTag 0xfe (@0x816f9c58)
  LuExt Size 0x00000038   SrbExt Size 0x00000188
    SG List Size - Small 17   Large 0
    Emergency  - SrbData 0x816f9830  Blocked List @0x816f9e94
    CommonBuff - Size: 0x00006000    PA: 0x0000000001702000  VA: 0x81702000
    Ke Objects - Int1: 0x8175ba50    Int2: 0x00000000        Dma: 0x816f9340
    Lookaside  - SrbData @ 0x816f9e40 SgList @0x00000000  Remove: @0x00000000
    Resources  - Raw: 0x817ba190     Translated: 0x81709678
    Port Config 8177fde8
    DeviceMap Handles: Port 0000009c    Busses e12d7b38
  Interrupt Data @0x816f9ce4:
    Flags (0x00000000):
    Ready LUN 0x00000000   Wmi Events 0x00000000
    Completed Request List (@0x816f9ce8): 0 entries
    LUN 816ea0e8 @ (  0,  1,  0) c ev  pnp(00/ff) pow(0 ,0) DevObj 816ea030
```

Here is an example of the **!scsikd.scsiext** display, where the SCSI port extension has been specified by a physical device object (PDO); this can be obtained from the **DevObj** field or **LUN** field in the **!minipkd.adapters** display:

```
kd> !scsikd.scsiext 816ea030
Scsiport physical device extension at address 816ea0e8
Common Extension:
  Initialized
  DO 0x816ea030  LowerObject 0x816f9a40  SRB Flags 00000000
  Current Power (D0,S0)  Desired Power D-1 Idle 0x8176c780
  Current PnP state 0x0    Previous state 0xff
  DispatchTable f9aee180   UsePathCounts (P0, H0, C0)
Logical Unit Extension:
  Address (2, 0, 1, 0) Claimed  Enumerated Visible
  LuFlags (0x00000000):
  Retry 0x00          Key 0x00000000
  Lock 0x00000000  Pause 0x00000000   CurrentLock: 0x00000000
  HwLuExt 0x8177ce10  Adapter 0x816f9af8  Timeout 0xffffffff
  NextLun 0x00000000  ReadyLun 0x00000000
  Pending 0x00000000  Busy 0x00000000     Untagged 0x00000000
  Q Depth 000 (1628450047)   InquiryData 0x816ea206
  DeviceMap Keys: Target 0x0000d0   Lun 00000000
  Bypass SRB_DATA blocks 4 @ 816ea270   List 816ea810
  RS Irp 8177dd80  Srb @ 816eaa0c   MDL @ 816eaa4c
  Request List @0x816ea1f0 is empty
```

!scsikd.srbdata

12/5/2018 • 2 minutes to read • Edit Online

The **!scsikd.srbdata** extension displays the specified SRB_DATA tracking block.

```
!scsikd.srbdata Address
```

Parameters

Address
Specifies the address of an SRB_DATA tracking block.

DLL

Windows 2000	Scsikd.dll
Windows XP and later	Scsikd.dll

Additional Information

For more information, see SCSI Miniport Debugging.

!minipkd.help

12/5/2018 • 2 minutes to read • Edit Online

The **!minipkd.help** extension displays help text for the Minipkd.dll extension commands.

```
!minipkd.help
```

DLL

Windows 2000	Unavailable
Windows XP and later	Minipkd.dll

Additional Information

For more information, see SCSI Miniport Debugging.

Remarks

If an error message similar to the following appears, it indicates that the symbol path is incorrect and does not point to the correct version of the Scsiport.sys symbols.

```
minipkd error (0) <path> ... \minipkd\minipkd.c @ line 435
```

Use the **.sympath (Set Symbol Path)** command to display the current path and change the path. Use the **.reload (Reload Module)** command to reload symbols from the current path.

!minipkd.adapter

12/5/2018 • 2 minutes to read • Edit Online

The **!minipkd.adapter** extension displays information about the specified adapter.

```
!minipkd.adapter Address
```

Parameters

Address
Specifies the address of an adapter.

DLL

Windows 2000	Unavailable
Windows XP and later	Minipkd.dll

Additional Information

For more information, see SCSI Miniport Debugging.

Remarks

The address of an adapter can be found in the **DevExt** field of the **!minipkd.adapters** display.

!minipkd.adapters

12/5/2018 • 2 minutes to read • Edit Online

The **!minipkd.adapters** extension displays all of the adapters that work with the SCSI Port driver that have been identified in the system, and the individual devices associated with each adapter.

```
!minipkd.adapters
```

DLL

Windows 2000	Unavailable
Windows XP and later	Minipkd.dll

Additional Information

For more information, see SCSI Miniport Debugging.

Remarks

The display includes the driver name, the device object address, and the device extension address for each adapter. The display for each adapter also includes a list of each device on the adapter. The display for each device includes the device extension address, the SCSI address, the device object address, and some flags for the device. Information about the Plug and Play state and the power state is also included.

The flags in the display are explained in the following table:

FLAG	MEANING
c	Claimed. Indicates that the device has a driver on it.
m	Missing. Indicates that the device was present on the bus in a prior scan but was not present during the latest scan.
e	Enumerated. Indicates that the device has been reported to the Plug and Play manager.
v	Visible. Indicates that the device has been enumerated by the system. This flag is more significant when it is not present for a device.
p	Paging. Indicates that the device is in the paging path.
d	Dump. Indicates that the device is in the crash dump path and will be used for a crash dump.

FLAG	MEANING
h	Hibernate. Indicates that the device is hibernating.

Here is an example of the **!minipkd.adapters** display:

```
0: kd> !minipkd.adapters
Adapter \Driver\lp6nds35     DO 86334a70         DevExt 86334b28
Adapter \Driver\adpu160m     DO 8633da70         DevExt 8633db28
 LUN 862e60f8 @(0,0,0) c ev     pnp(00/ff) pow(0,0) DevObj 862e6040
 LUN 863530f8 @(0,1,0) c ev p d pnp(00/ff) pow(0,0) DevObj 86353040
 LUN 862e50f8 @(0,2,0) c ev     pnp(00/ff) pow(0,0) DevObj 862e5040
 LUN 863520f8 @(0,6,0)   ev     pnp(00/ff) pow(0,0) DevObj 86352040
Adapter \Driver\adpu160m     DO 86376040         DevExt 863760f8
```

An error message similar to the following indicates that either the symbol path is incorrect and does not point to the correct version of the Scsiport.sys symbols, or that Windows has not identified any adapters that work with the SCSI Port driver.

```
minipkd error (0) <path> ... \minipkd\minipkd.c @ line 435
```

If the **!minipkd.help** extension command returns help information successfully, the SCSI Port symbols are correct.

!minipkd.exports

12/5/2018 • 2 minutes to read • Edit Online

The **!minipkd.exports** extension displays the addresses of the miniport exports for the specified adapter.

```
!minipkd.exports Adapter
```

Parameters

Adapter
Specifies the address of an adapter.

DLL

Windows 2000	Unavailable
Windows XP and later	Minipkd.dll

Additional Information

For more information, see SCSI Miniport Debugging.

!minipkd.lun

12/5/2018 • 2 minutes to read • Edit Online

The **!minipkd.lun** extension displays detailed information about the specified Logical Unit Extension (LUN).

```
!minipkd.lun LUN
!minipkd.lun Device
```

Parameters

LUN
Specifies the address of the LUN.

Device
Specifies the physical device object (PDO) for the LUN.

DLL

Windows 2000	Unavailable
Windows XP and later	Minipkd.dll

Additional Information

For more information, see SCSI Miniport Debugging.

Remarks

A LUN is typically referred to as a *device*. Thus, this extension displays information about a device on an adapter.

The LUN can be specified either by its address (which can be found in the **LUN** field of the **!minipkd.adapters** display), or by its physical device object (which can be found in the **DevObj** field of the **!minipkd.adapters** display).

!minipkd.portconfig

12/5/2018 • 2 minutes to read • Edit Online

The **!minipkd.portconfig** extension displays information about the specified PORT_CONFIGURATION_INFORMATION data structure.

```
!minipkd.portconfig PortConfig
```

Parameters

PortConfig
Specifies the address of a PORT_CONFIGURATION_INFORMATION data structure.

DLL

Windows 2000	Unavailable
Windows XP and later	Minipkd.dll

Additional Information

For more information, see SCSI Miniport Debugging.

Remarks

The *PortConfig* address can be found in the **Port Config Info** field of the **!minipkd.adapter** display.

!minipkd.req

12/5/2018 • 2 minutes to read • Edit Online

The **!minipkd.req** extension displays information about all of the currently active requests on the specified adapter or device.

```
!minipkd.req Adapter
!minipkd.req Device
```

Parameters

Adapter
Specifies the address of the adapter.

Device
Specifies the physical device object (PDO) for the Logical Unit Extension (LUN) device.

DLL

Windows 2000	Unavailable
Windows XP and later	Minipkd.dll

Additional Information

For more information, see SCSI Miniport Debugging.

Remarks

The PDO for a LUN can be found in the **DevObj** field of the **!minipkd.adapters** display.

!minipkd.srb

12/5/2018 • 2 minutes to read • Edit Online

The **!minipkd.srb** extension displays the specified SCSI request block (SRB) data structure.

```
!minipkd.srb SRB
```

Parameters

SRB
Specifies the address of an SRB.

DLL

Windows 2000	Unavailable
Windows XP and later	Minipkd.dll

Additional Information

For more information, see SCSI Miniport Debugging.

Remarks

The addresses of all currently active requests can be found in the *SRB* fields of the output from the **!minipkd.req** command.

This extension displays the status of the SRB, the driver it is addressed to, the SCSI that issued the SRB and its address, and a hexadecimal flag value. If 0x10000 is set in the flag value, this request is currently in the miniport.

Windows Driver Framework Extensions (Wdfkd.dll)

12/5/2018 • 2 minutes to read • Edit Online

Extension commands that are useful for debugging drivers built with the Kernel-Mode Driver Framework (KMDF) or version 2 of the User-Mode Driver Framework (UMDF 2) are implemented in Wdfkd.dll.

These extensions can be used on Microsoft Windows XP and later operating systems. Some extensions have additional restrictions; these restrictions are noted on the individual reference pages.

Note When you create a new KMDF or UMDF driver, you must select a driver name that has 32 characters or less. This length limit is defined in wdfglobals.h. If your driver name exceeds the maximum length, your driver will fail to load.

For more information about how to use these extensions, see Kernel-Mode Driver Framework Debugging.

!wdfkd.help

12/5/2018 • 2 minutes to read • Edit Online

The **!wdfkd.help** extension displays help information about all Wdfkd.dll extension commands.

```
!wdfkd.help
```

DLL

Wdfkd.dll

Additional Information

The **!wdfkd.help** extension is equivalent to the **!wdfkd.wdfhelp** extension.

For more information about debugging framework-based drivers, see Kernel-Mode Driver Framework Debugging.

!wdfkd.wdfchildlist

12/5/2018 • 2 minutes to read • Edit Online

The **!wdfkd.wdfchildlist** extension displays a child list's state and information about all of the device identification descriptions that are in the child list.

```
!wdfkd.wdfchildlist Handle
```

Parameters

Handle
A WDFCHILDLIST-typed handle to the child list.

DLL

Wdfkd.dll

Frameworks

KMDF 1

Additional Information

For more information, see Kernel-Mode Driver Framework Debugging.

Remarks

The following example shows a **!wdfkd.wdfchildlist** display.

```
kd> !wdfchildlist 0x7cc090c8

## Dumping WDFCHILDLIST 0x7cc090c8
---------------------------------------
owning !WDFDEVICE 0x7ca7b1c0
ID description size 0x8

State:
-----
List is unlocked, changes will be applied immediately
No scans or enumerations are active on the list

Descriptions:
------------
 PDO !WDFDEVICE 0x7cad31c8, ID description 0x83ac4ff4
 +Device WDM !devobj 0x81fb00e8, WDF pnp state WdfDevStatePnpStarted (0x119)
 +Device found in last scan

No pending insertions are in the list.

Callbacks:
---------
 EvtChildListCreateDevice:  wdfrawbusenumtest!RawBus_RawPdo_Create (f22263b0)
```

!wdfkd.wdfcollection

12/5/2018 • 2 minutes to read • Edit Online

The **!wdfkd.wdfcollection** extension displays all of the objects that are stored in a WDFCOLLECTION structure.

```
!wdfkd.wdfcollection Handle
```

Parameters

Handle
A WDFCOLLECTION-typed handle to the structure.

DLL

Wdfkd.dll

Frameworks

KMDF 1, UMDF 2

Additional Information

For more information, see Kernel-Mode Driver Framework Debugging.

!wdfkd.wdfcommonbuffer

12/5/2018 • 2 minutes to read • Edit Online

The **!wdfkd.wdfcommonbuffer** extension displays information about a WDF common buffer object.

```
!wdfkd.wdfcommonbuffer Handle
```

Parameters

Handle
A handle to a framework common buffer object (WDFCOMMONBUFFER).

DLL

Wdfkd.dll

Frameworks

KMDF 1, UMDF 2

Additional Information

For more information, see Kernel-Mode Driver Framework Debugging.

!wdfkd.wdfcrashdump

6/25/2019 • 2 minutes to read • Edit Online

The **!wdfkd.wdfcrashdump** extension displays error log information and other crash dump information from a minidump file, if the data is present.

KMDF

```
!wdfkd.wdfcrashdump [InfoType]
```

UMDF

```
!wdfkd.wdfcrashdump [DriverName.dll][-d | -f | -m]
```

Parameters

InfoType
Specifies the kind of information to display. *InfoType* is optional and can be any one of the following values:

log
Displays error log information, if available in the crash dump file. This is the default value.

loader
Displays the minidump's dynamic-bound drivers.

DriverName.dll
Specifies the name of a UMDF driver. You must include the .dll file suffix. If this optional parameter is omitted, output includes metadata, the loaded module list, and available logs.

-d
Displays only the driver logs.

-f
Displays only the framework logs.

-m
Merges framework and driver logs in their recorded order.

DLL

Wdfkd.dll

Frameworks

KMDF

UMDF 2.15

Remarks

This example shows how to use **!wdfkd.wdfcrashdump** to view information about KMDF drivers. If you specify **loader** for *InfoType*, the output includes dynamic-bound drivers in the minidump file.

```
0: kd> !wdfcrashdump loader
Retrieving crashdump loader information...
## Local buffer 0x002B4D00, bufferSize 720
-------------------------------------------
  ImageName      Version    FxGlobals

  Wdf01000       v1.9(6902)
  msisadrv       v1.9(6913) 0x84deb260
  vdrvroot       v1.9(6913) 0x860e8260
  storflt        v1.5(6000) 0x861dfe90
  cdrom          v1.9(6913) 0x84dca008
  intelppm       v1.9(6913) 0x864704a8
  HDAudBus       v1.7(6001) 0x86101c98
  1394ohci       v1.7(6001) 0x8610d2e8
  CompositeBus   v1.9(6913) 0x86505b98
  ObjTestClassExt v1.9(6902) 0x865b7f00
  mqfilter       v1.9(6902) 0x865b8008
  mqueue         v1.9(6902) 0x865b6910
  umbus          v1.9(6913) 0x8618aea0
  monitor        v1.9(6913) 0x86aac1d8
  PEAUTH         v1.5(6000) 0x854e5350
-------------------------------------------
```

This example shows how to use **!wdfkd.wdfcrashdump** to view information about UMDF drivers. If you issue **!wdfkd.wdfcrashdump** with no parameters, the output includes the driver that caused the crash and a list of all loaded drivers in the host process that failed. You can click on drivers in this list that have associated logs.

```
0:001> !wdfkd.wdfcrashdump
Opening minidump at location C:\temp\WudfHost_ext__1312.dmp

Faulting driver: wpptest.dll
Failure type: Unhandled Exception (WUDFUnhandledException)
Faulting thread ID: 2840

Listing all drivers loaded in this host process at the time of the failure:

  ServiceName
  wpptest
  CoverageCx0102
  coverage
  WUDFVhidmini
  ToastMon
  WUDFOsrUsbFilter
```

In the example above, output includes failure type, which is the event type in the WER report. Here, it can be **WUDFVerifierFailure** or **WUDFUnhandledException**. For more information, see Accessing UMDF Metadata in WER Reports. The output for UMDF includes an error code, if event type is **WUDFVerifierFailure**.

To display the framework's error log records from a complete memory dump, a kernel memory dump, or a live kernel-mode target, you can also try the **!wdfkd.wdflogdump** extension.

Additional Information

For information about enabling the inflight trace recorder for your driver, see Using Inflight Trace Recorder (IFR) in KMDF and UMDF 2 Drivers. For more information about debugging WDF drivers, see Debugging WDF Drivers. For information about KMDF debugging, see Kernel-Mode Driver Framework Debugging.

See also

!wdfkd.wdflogdump

!wdfkd.wdfsettraceprefix

!wdfkd.wdfdevext

12/5/2018 • 2 minutes to read • Edit Online

The **!wdfkd.wdfdevext** extension displays information that is associated with the **DeviceExtension** member of a Microsoft Windows Driver Model (WDM) DEVICE_OBJECT structure.

```
!wdfkd.wdfdevext DeviceExtension
```

Parameters

DeviceExtension
A pointer to a device extension.

DLL

Wdfkd.dll

Frameworks

KMDF 1, UMDF 1, UMDF 2

Additional Information

For more information, see Kernel-Mode Driver Framework Debugging.

Remarks

Here is an example for HdAudBus.sys, which is a KMDF driver. Use **!devnode** to find a device node that has HdAudBus as its function driver. Take the physical device object (PDO) from the output and pass it to **!devstack**. Take the device extension address from the output of **!devstack** and pass it to **!wdfdevext**.

```
0: kd> !devnode 0 1 hdaudbus
Dumping IopRootDeviceNode (= 0xffffe000002cfd30)
DevNode 0xffffe000009b7a50 for PDO 0xffffe00000226880
  InstancePath is "PCI\VEN_8086&DEV_293E&SUBSYS_2819103C&REV_02\3&33fd14ca&0&D8"
  ServiceName is "HDAudBus"
  ...
0: kd> !devstack 0xffffe00000226880
  !DevObj           !DrvObj            !DevExt           ObjectName
  ffffe00001351e20  \Driver\HDAudBus   ffffe000009a3c00
> ffffe00000226880  \Driver\pci        ffffe000002269d0  NTPNP_PCI0009
!DevNode ffffe000009b7a50 :
  DeviceInst is "PCI\VEN_8086&DEV_293E&SUBSYS_2819103C&REV_02\3&33fd14ca&0&D8"
  ServiceName is "HDAudBus"
0: kd> *
0: kd> !wdfdevext ffffe000009a3c00
Device context is 0xffffe000009a3c00
    context:  dt 0xffffe000009a3c00 HDAudBus!HDAudioDeviceExtension (size is 0xa8 bytes)
    EvtCleanupCallback fffff80001f35950 HDAudBus!HdAudBusEvtDeviceCleanupCallback

!wdfdevice 0x00001fffff65c6e8
!wdfobject 0xffffe000009a3910
```

Here is an example for Wudfrd.sys, which is the function driver for the kernel-mode portion of a UMDF 2 driver stack. Use **!devnode** to find a device node that has Wudfrd as its function driver. Take the physical device object (PDO) from the output and pass it to **!devstack**. Take the device extension address from the output of **!devstack**

and pass it to **!wdfdevext**.

```
0: kd> !devnode 0 1 wudfrd
Dumping IopRootDeviceNode (= 0xffffe000002cfd30)
DevNode 0xffffe00000a1e530 for PDO 0xffffe00000b15b00
  InstancePath is "ROOT\SAMPLE\0001"
  ServiceName is "WUDFRd"
  ...
0: kd> !devstack 0xffffe00000b15b00
  !DevObj           !DrvObj            !DevExt           ObjectName
  ffffe00000c11040  \Driver\WUDFRd     ffffe00000c11190
> ffffe00000b15b00  \Driver\PnpManager 00000000  00000052
!DevNode ffffe00000a1e530 :
  DeviceInst is "ROOT\SAMPLE\0001"
  ServiceName is "WUDFRd"
0: kd> *
0: kd> !wdfdevext ffffe00000c11190
## Device context is 0xffffe00000c11190

##  UMDF Device Instances for this Redirector extension

  DriverManagerProcess: 0xffffe00003470500

  ImageName              Ver   DevStack           HostProcess        DeviceID
  MyUmdf2Driver.dll      v2.0  0x000000a5a3ab5f70 0xffffe00000c32900  \Device\00000052
```

!wdfkd.wdfdevice

12/5/2018 • 2 minutes to read • Edit Online

The **!wdfkd.wdfdevice** extension displays information that is associated with a WDFDEVICE-typed object handle.

```
!wdfkd.wdfdevice Handle [Flags]
```

Parameters

Handle
A handle to a WDFDEVICE-typed object.

Flags
Optional. The kind of information to display. *Flags* can be any combination of the following bits:

Bit 0 (0x1)
The display will include verbose information about the device, such as the associated WDFCHILDLIST-typed handles, synchronization scope, and execution level.

Bit 1 (0x2)
The display will include detailed power state information.

Bit 2 (0x4)
The display will include detailed power policy state information.

Bit 3 (0x8)
The display will include detailed Plug and Play (PnP) state information.

Bit 4 (0x10)
The display will include the device object's callback functions.

DLL

Wdfkd.dll

Frameworks

KMDF 1, UMDF 2

Additional Information

For more information, see Kernel-Mode Driver Framework Debugging.

Remarks

The following example uses the **!wdfkd.wdfdevice** extension on a WDFDEVICE handle that represents a physical device object (PDO), without specifying any flags.

```
kd> !wdfdevice 0x7cad31c8

# Dumping WDFDEVICE 0x7cad31c8
================================

WDM PDEVICE_OBJECTs:  self 81fb00e8

Pnp state:  119 ( WdfDevStatePnpStarted )
Power state:  31f ( WdfDevStatePowerDx )
Power Pol state:  508 ( WdfDevStatePwrPolWaitingUnarmed )

Parent WDFDEVICE 7ca7b1c0
Parent states:
   Pnp state:  119 ( WdfDevStatePnpStarted )
   Power state:  307 ( WdfDevStatePowerD0 )
   Power Pol state:  565 ( WdfDevStatePwrPolStarted )

No pended pnp or power irps
Device is the power policy owner for the stack
```

The following example displays the same device object as the preceding example, but this time with a flag value of 0xF. This flag value, a combination of the bits 0x1, 0x2, 0x4, and 0x8, causes the display to include verbose device information, power state information, power policy state information, and PnP state information.

```
kd> !wdfdevice 0x7cad31c8 f

# Dumping WDFDEVICE 0x7cad31c8
================================

WDM PDEVICE_OBJECTs:  self 81fb00e8

Pnp state:  119 ( WdfDevStatePnpStarted )
Power state:  31f ( WdfDevStatePowerDx )
Power Pol state:  508 ( WdfDevStatePwrPolWaitingUnarmed )

Parent WDFDEVICE 7ca7b1c0
Parent states:
   Pnp state:  119 ( WdfDevStatePnpStarted )
   Power state:  307 ( WdfDevStatePowerD0 )
   Power Pol state:  565 ( WdfDevStatePwrPolStarted )

No pended pnp or power irps
Device is the power policy owner for the stack

Pnp state history:
[0] WdfDevStatePnpObjectCreated (0x100)
[1] WdfDevStatePnpInit (0x105)
[2] WdfDevStatePnpInitStarting (0x106)
[3] WdfDevStatePnpHardwareAvailable (0x108)
[4] WdfDevStatePnpEnableInterfaces (0x109)
[5] WdfDevStatePnpStarted (0x119)

Power state history:
[0] WdfDevStatePowerD0StartingConnectInterrupt (0x310)
[1] WdfDevStatePowerD0StartingDmaEnable (0x311)
[2] WdfDevStatePowerD0StartingStartSelfManagedIo (0x312)
[3] WdfDevStatePowerDecideD0State (0x313)
[4] WdfDevStatePowerD0BusWakeOwner (0x309)
[5] WdfDevStatePowerGotoDx (0x31a)
[6] WdfDevStatePowerGotoDxIoStopped (0x31c)
[7] WdfDevStatePowerDx (0x31f)

Power policy state history:
[0] WdfDevStatePwrPolStarting (0x501)
[1] WdfDevStatePwrPolStartingSucceeded (0x502)
[2] WdfDevStatePwrPolStartingDecideS0Wake (0x504)
[3] WdfDevStatePwrPolStartedIdleCapable (0x505)
[4] WdfDevStatePwrPolTimerExpiredNoWake (0x506)
[5] WdfDevStatePwrPolTimerExpiredNoWakeCompletePowerDown (0x507)
[6] WdfDevStatePwrPolWaitingUnarmedQueryIdle (0x509)
[7] WdfDevStatePwrPolWaitingUnarmed (0x508)

WDFCHILDLIST Handles:
 !WDFCHILDLIST 0x7ce710c8

SyncronizationScope is WdfSynchronizationScopeNone
ExecutionLevel is WdfExecutionLevelDispatch
```

!wdfkd.wdfdeviceinterrupts

12/5/2018 • 2 minutes to read • Edit Online

The **!wdfkd.wdfdeviceinterrupts** extension displays all the interrupt objects for a specified device handle.

```
!wdfkd.wdfdeviceinterrupts Handle
```

Parameters

Handle
A handle to a WDFDEVICE-typed object.

DLL

Wdfkd.dll

Frameworks

KMDF 1, UMDF 2

Additional Information

For more information, see Kernel-Mode Driver Framework Debugging.

!wdfkd.wdfdevicequeues

12/5/2018 • 2 minutes to read • Edit Online

The **!wdfkd.wdfdevicequeues** extension displays information about all of the framework queue objects that belong to a specified device.

```
!wdfkd.wdfdevicequeues Handle
```

Parameters

Handle
A handle to a WDFDEVICE-typed object.

DLL

Wdfkd.dll

Frameworks

KMDF 1, UMDF 2

Additional Information

For more information, see Kernel-Mode Driver Framework Debugging and **!wdfkd.wdfqueue**.

Remarks

The following example shows the display from the **!wdfkd.wdfdevicequeues** extension.

```
kd> !wdfdevicequeues 0x7cad31c8

# Dumping queues of WDFDEVICE 0x7cad31c8
====================================
## Number of queues: 3
---------------------------------
Queue: 1 (!wdfqueue 0x7d67d1e8)
    Manual, Not power-managed, PowerOn, Can accept, Can dispatch, ExecutionLevelDispatch,
SynchronizationScopeNone
    Number of driver owned requests: 0
    Number of waiting requests: 0


## This is WDF internal queue for create requests.
---------------------------------
Queue: 2 (!wdfqueue 0x7ce7d1e8)
    Parallel, Power-managed, PowerOff, Can accept, Can dispatch, ExecutionLevelDispatch,
SynchronizationScopeNone
    Number of driver owned requests: 0
    Number of waiting requests: 0


##     EvtIoDefault: (0xf221fad0) wdfrawbusenumtest!EvtIoQueueDefault
---------------------------------
Queue: 3 (!wdfqueue 0x7cd671e8)
    Parallel, Power-managed, PowerOff, Can accept, Can dispatch, ExecutionLevelDispatch,
SynchronizationScopeNone
    Number of driver owned requests: 0
    Number of waiting requests: 0


    EvtIoDeviceControl: (0xf2226ac0) wdfrawbusenumtest!RawBus_RawPdo_EvtDeviceControl
```

!wdfkd.wdfdmaenabler

12/5/2018 • 2 minutes to read • Edit Online

The **!wdfkd.wdfdmaenabler** extension displays information about a WDF direct memory access (DMA) object, and its transaction and common buffer objects.

```
!wdfkd.wdfdmaenabler Handle
```

Parameters

Handle
A handle to a framework DMA enabler object (WDFDMAENABLER).

DLL

Wdfkd.dll

Frameworks

KMDF 1

Additional Information

For more information, see Kernel-Mode Driver Framework Debugging.

!wdfkd.wdfdmaenablers

12/5/2018 • 2 minutes to read • Edit Online

The **!wdfkd.wdfdmaenablers** extension displays all WDF direct memory access (DMA) objects associated with a specified device object. It also displays their associated transaction and common buffer objects.

```
!wdfkd.wdfdmaenablers Handle
```

Parameters

Handle
A handle to a framework device object (WDFDEVICE).

DLL

Wdfkd.dll

Frameworks

KMDF 1

Additional Information

For more information, see Kernel-Mode Driver Framework Debugging.

!wdfkd.wdfdmatransaction

12/5/2018 • 2 minutes to read • Edit Online

The **!wdfkd.wdfdmatransaction** extension displays information about a WDF direct memory access (DMA) transaction object.

```
!wdfkd.wdfdmatransaction Handle
```

Parameters

Handle
A handle to a framework DMA transaction object (WDFDMATRANSACTION).

DLL

Wdfkd.dll

Frameworks

KMDF 1

Additional Information

For more information, see Kernel-Mode Driver Framework Debugging.

!wdfkd.wdfdriverinfo

12/5/2018 • 2 minutes to read • Edit Online

The **!wdfkd.wdfdriverinfo** extension displays information about the specified driver, including its device tree, the version of the Kernel-Mode Driver Framework (KMDF) library that the driver was compiled with, and a list of the framework device objects that the driver created.

```
!wdfkd.wdfdriverinfo [DriverName [Flags]]
```

Parameters

DriverName
Optional. The name of the driver. *DriverName* must not include the .sys file name extension.

Flags
Optional. Flags that specify the kind of information to display. *Flags* can be any combination of the following bits:

Bit 0 (0x1)
The display will include verifier settings for the driver, and will also include a count of WDF objects. This flag can be combined with bit 6 (0x40) to display internal objects.

Bit 4 (0x10)
The display will include the KMDF handle hierarchy for the driver.

Bit 5 (0x20)
The display will include context and callback function information for each handle. This flag is valid only when bit 4 (0x10) is set.

Bit 6 (0x40)
The display will include additional information for each handle. This flag is valid only when bit 4 (0x10) is set. This flag can be combined with bit 0 (0x1) to display internal objects.

Bit 7 (0x80)
The handle information will be displayed in a more compact format.

Bit 8 (0x100)
The display will left align internal type information. This flag is valid only when bit 4 (0x10) is set.

Bit 9 (0x200)
The display will include handles that the driver potentially leaked. KMDF version 1.1 and later support this flag. This flag is valid only when bit 4 (0x10) is set.

Bit 10 (0x400)
The display will include the device tree in verbose form.

DLL

Wdfkd.dll

Frameworks

KMDF 1, UMDF 2

Additional Information

For more information, see Kernel-Mode Driver Framework Debugging.

Remarks

If you omit the *DriverName* parameter, the default driver is used. You can display the default driver by using the **!wdfkd.wdfgetdriver** extension; you can set the default driver by using the **!wdfkd.wdfsetdriver** extension.

The following example shows the display from the **!wdfkd.wdfdriverinfo** extension.

```
## kd> !wdfdriverinfo wdfrawbusenumtest
---------------------------------
Default driver image name:   wdfrawbusenumtest
WDF library image name:      Wdf01000
 FxDriverGlobals  0x83b7af18
 WdfBindInfo      0xf22250ec
##    Version        v1.5 build(1234)
---------------------------------
WDFDRIVER: 0x7cbc90d0

    !WDFDEVICE 0x7ca7b1c0
            context:  dt 0x83584ff8 ROOT_CONTEXT (size is 0x1 bytes)
             <no associated attribute callbacks>

    !WDFDEVICE 0x7cad31c8
            context:  dt 0x8352cff0 RAW_PDO_CONTEXT (size is 0xc bytes)
             <no associated attribute callbacks>
```

!wdfkd.wdfextendwatchdog

12/5/2018 • 2 minutes to read • Edit Online

The **!wdfkd.wdfextendwatchdog** extension extends the time-out period (from 10 minutes to 24 hours) of the framework's watchdog timer during power transitions.

```
!wdfkd.wdfextendwatchdog Handle [Extend]
```

Parameters

Handle
A handle to a WDFDEVICE-typed object.

Extend
Optional. A value that indicates whether to enable or disable extension of the time-out period. If *Extend* is 0, extension is disabled, and the time-out period is 10 minutes. If *Extend* is 1, extension is enabled and the time-out period is 24 hours. The default value is 1.

DLL

Wdfkd.dll

Frameworks

KMDF 1

Additional Information

For more information, see Kernel-Mode Driver Framework Debugging.

Remarks

The framework starts an internal watchdog timer every time it calls a power policy or power event callback function for a driver that is not power pageable (that is, the DO_POWER_PAGABLE bit is clear). If the callback function causes paging I/O and therefore blocks, the operating system hangs because no paging device is available to service the request.

If the time-out period elapses, the framework issues bug check 0x10D (WDF_VIOLATION). For details, see **Bug Check 0x10D**.

!wdfkd.wdffindobjects

12/5/2018 • 2 minutes to read • Edit Online

The **!wdfkd.wdffindobjects** extension searches memory for WDF objects.

```
!wdfkd.wdffindobjects [StartAddress [Flags]]
```

Parameters

StartAddress
Optional. Specifies the address at which the search must begin. If this is omitted, the search will begin from where the most recent **!wdfkd.wdffindobjects** search ended.

Flags
Optional. Specifies the kind of information to display. *Flags* can be any combination of the following bits. The default value is 0x0. *Flags* cannot be used unless *StartAddress* is specified.

Bit 0 (0x1)
Displays verbose output.

Bit 1 (0x2)
Displays internal type information for each handle.

DLL

Wdfkd.dll

Frameworks

KMDF 1, UMDF 2

Additional Information

For more information, see Kernel-Mode Driver Framework Debugging.

Remarks

The following examples show the output of the **!wdfkd.wdffindobjects** extension. The 0x1 flag is set in the second example.

```
1: kd> !wdffindobjects 0xfffffa600211b668
  Address             Value               Object
  ------------------  ------------------  ------------------
  0xfffffa600211b668  0x0000000000000008
  0xfffffa600211b670  0xfffffa8002e7b1f0  !WDFREQUEST 0x0000057ffd184e08
  0xfffffa600211b678  0x0000000000000004
  0xfffffa600211b680  0x0000000000000001
  0xfffffa600211b688  0xfffffa8006aa3640  !WDFUSBPIPE 0x0000057ff955c9b8
  0xfffffa600211b690  0x0000000000000000
  0xfffffa600211b698  0xfffff80001e61f78
  0xfffffa600211b6a0  0x0000000000000010
  0xfffffa600211b6a8  0x0000000000010286
  0xfffffa600211b6b0  0xfffffa600211b6c0
  0xfffffa600211b6b8  0x0000000000000000
  0xfffffa600211b6c0  0xfffffa8006aa3640  !WDFUSBPIPE 0x0000057ff955c9b8
  0xfffffa600211b6c8  0x0000057ffd184e08  !WDFREQUEST 0x0000057ffd184e08
  0xfffffa600211b6d0  0x0000000000000000
  0xfffffa600211b6d8  0x0000057ffc51ea18  !WDFMEMORY 0x0000057ffc51ea18
  0xfffffa600211b6e0  0x0000000000000000

1: kd> !wdffindobjects 0xfffffa600211b668 1
  Address             Value               Type    Object
  ------------------  ------------------  ------  ------------------
  0xfffffa600211b668  0x0000000000000008
  0xfffffa600211b670  0xfffffa8002e7b1f0  Object  !WDFREQUEST 0x0000057ffd184e08
  0xfffffa600211b678  0x0000000000000004
  0xfffffa600211b680  0x0000000000000001
  0xfffffa600211b688  0xfffffa8006aa3640  Object  !WDFUSBPIPE 0x0000057ff955c9b8
  0xfffffa600211b690  0x0000000000000000
  0xfffffa600211b698  0xfffff80001e61f78
  0xfffffa600211b6a0  0x0000000000000010
  0xfffffa600211b6a8  0x0000000000010286
  0xfffffa600211b6b0  0xfffffa600211b6c0
  0xfffffa600211b6b8  0x0000000000000000
  0xfffffa600211b6c0  0xfffffa8006aa3640  Object  !WDFUSBPIPE 0x0000057ff955c9b8
  0xfffffa600211b6c8  0x0000057ffd184e08  Handle  !WDFREQUEST 0x0000057ffd184e08
  0xfffffa600211b6d0  0x0000000000000000
  0xfffffa600211b6d8  0x0000057ffc51ea18  Handle  !WDFMEMORY 0x0000057ffc51ea18
  0xfffffa600211b6e0  0x0000000000000000
```

!wdfkd.wdfforwardprogress

12/5/2018 • 2 minutes to read • Edit Online

The **!wdfkd.wdfforwardprogress** extension displays information about the forward progress of a specified framework queue object.

```
!wdfkd.wdfforwardprogress Handle
```

Parameters

Handle
A handle to a framework queue object.

DLL

Wdfkd.dll

Frameworks

KMDF 1

Additional Information

For more information about how to debug Kernel-Mode Driver Framework (KMDF) drivers, see Kernel-Mode Driver Framework Debugging.

Remarks

This extension will succeed only if the specified framework queue object is configured to support forward progress. If this extension is used with other objects, an error message will be displayed.

The following example shows the display from a **!wdfkd.wdfforwardprogress** extension.

```
kd> !wdfkd.wdfforwardprogress 0x79af3250

# Dumping forward progress fields for WDFQUEUE 0x79af3250
=================================================
    ForwardProgressReservedPolicy: UseExamine (0x2)

    Total reserved requests: 44

    Number of available reserved requests in list: 41
            !WDFREQUEST 0x7bcaf4c0 (Reserved) !IRP 0x00000000
            !WDFREQUEST 0x7bc67eb0 (Reserved) !IRP 0x00000000
            !WDFREQUEST 0x7bccf678 (Reserved) !IRP 0x00000000
            !WDFREQUEST 0x7bb6ce40 (Reserved) !IRP 0x00000000
            !WDFREQUEST 0x7be30a58 (Reserved) !IRP 0x00000000
            !WDFREQUEST 0x79af37d0 (Reserved) !IRP 0x00000000
            !WDFREQUEST 0x7bc7f428 (Reserved) !IRP 0x00000000
            !WDFREQUEST 0x7bbd40f0 (Reserved) !IRP 0x00000000
            !WDFREQUEST 0x7bd333a8 (Reserved) !IRP 0x00000000
            !WDFREQUEST 0x7bd241d8 (Reserved) !IRP 0x00000000
            !WDFREQUEST 0x7bd594e0 (Reserved) !IRP 0x00000000
            !WDFREQUEST 0x7bd80d10 (Reserved) !IRP 0x00000000
            !WDFREQUEST 0x78ea2d50 (Reserved) !IRP 0x00000000
            !WDFREQUEST 0x792020f0 (Reserved) !IRP 0x00000000
            !WDFREQUEST 0x7bc37258 (Reserved) !IRP 0x00000000
            !WDFREQUEST 0x7bbc1fb0 (Reserved) !IRP 0x00000000
            !WDFREQUEST 0x7bbc4fb0 (Reserved) !IRP 0x00000000
            !WDFREQUEST 0x7be0cb80 (Reserved) !IRP 0x00000000
            !WDFREQUEST 0x7bc84890 (Reserved) !IRP 0x00000000
            !WDFREQUEST 0x78acbd18 (Reserved) !IRP 0x00000000
            !WDFREQUEST 0x7bcf1ad8 (Reserved) !IRP 0x00000000
            !WDFREQUEST 0x7bead540 (Reserved) !IRP 0x00000000
            !WDFREQUEST 0x7922c0f0 (Reserved) !IRP 0x00000000
            !WDFREQUEST 0x7a34a0f0 (Reserved) !IRP 0x00000000
            !WDFREQUEST 0x625195d0 (Reserved) !IRP 0x00000000
            !WDFREQUEST 0x7bc33640 (Reserved) !IRP 0x00000000
            !WDFREQUEST 0x7bba9f28 (Reserved) !IRP 0x00000000
            !WDFREQUEST 0x7bba44c8 (Reserved) !IRP 0x00000000
            !WDFREQUEST 0x7bb77cd8 (Reserved) !IRP 0x00000000
            !WDFREQUEST 0x7a2b89a8 (Reserved) !IRP 0x00000000
            !WDFREQUEST 0x7a41ab88 (Reserved) !IRP 0x00000000
            !WDFREQUEST 0x7bc7cc88 (Reserved) !IRP 0x00000000
            !WDFREQUEST 0x7bd37180 (Reserved) !IRP 0x00000000
            !WDFREQUEST 0x7bca40f0 (Reserved) !IRP 0x00000000
            !WDFREQUEST 0x64b4af20 (Reserved) !IRP 0x00000000
            !WDFREQUEST 0x7bd01a40 (Reserved) !IRP 0x00000000
            !WDFREQUEST 0x7a25cfb0 (Reserved) !IRP 0x00000000
            !WDFREQUEST 0x7bba9330 (Reserved) !IRP 0x00000000
            !WDFREQUEST 0x7bd14a40 (Reserved) !IRP 0x00000000
            !WDFREQUEST 0x7bcc0210 (Reserved) !IRP 0x00000000
            !WDFREQUEST 0x7a54eb00 (Reserved) !IRP 0x00000000

    Number of reserved requests in use: 3
            !WDFREQUEST 0x7bf0ab80 (Reserved) !IRP 0x8438f008
            !WDFREQUEST 0x7bc53ca8 (Reserved) !IRP 0x875f59f0
            !WDFREQUEST 0x7bced8b8 (Reserved) !IRP 0x85c25348

    Number of undispatched IRP's in list: 0

    EvtIoReservedResourcesAllocate: (0x9a3f1b70) mqueue!EvtIoAllocateResourcesForReservedRequest
    EvtIoExamineIrp: (0x9a3f19d0) mqueue!EvtIoWdmIrpForForwardProgress
```

!wdfkd.wdfgetdriver

12/5/2018 • 2 minutes to read • Edit Online

The **!wdfkd.wdfgetdriver** extension displays the name of the current default driver.

```
!wdfkd.wdfgetdriver
```

DLL

Wdfkd.dll

Frameworks

KMDF 1, UMDF 2

Additional Information

For more information, see Kernel-Mode Driver Framework Debugging.

!wdfkd.wdfhandle

12/5/2018 • 2 minutes to read • Edit Online

The **!wdfkd.wdfhandle** extension displays information about a specified framework object handle, such as the handle type, object context pointers, and the underlying framework object pointer.

```
!wdfkd.wdfhandle Handle [Flags]
```

Parameters

Handle
A handle to a framework object.

Flags
Optional. Flags that specify the kind of information to display. *Flags* can be any combination of the following bits. The default value is 0x0.

Bit 4 (0x10)
The display will include the subtree of child objects for the specified handle.

Bit 5 (0x20)
The display will include context and callback function information for the specified handle. This flag is valid only when bit 4 (0x10) is set.

Bit 6 (0x40)
The display will include additional information for the specified handle. This flag is valid only when bit 4 (0x10) is set.

Bit 7 (0x80)
The handle information will be displayed in a more compact format.

Bit 8 (0x100)
The display will left align internal type information. This flag is valid only when bit 4 (0x10) is set.

DLL

Wdfkd.dll

Frameworks

KMDF 1, UMDF 2

Additional Information

For more information, see Kernel-Mode Driver Framework Debugging.

Remarks

The following example shows the output of the **!wdfhandle** extension with bit 4 set in the *Flags* parameter (so the output displays information about the child objects).

```
kd> !wdfhandle 0x7ca7b1c0 10

handle 0x7ca7b1c0, type is WDFDEVICE

Contexts:
    context:  dt 0x83584ff8 ROOT_CONTEXT (size is 0x1 bytes)
     <no associated attribute callbacks>

Child WDFHANDLEs of 0x7ca7b1c0:
    WDFDEVICE 0x7ca7b1c0
        WDFCMRESLIST 0x7ccfb058
        WDFCMRESLIST 0x7cadb058
        WDFCHILDLIST 0x7c72f0c8
        WDFCHILDLIST 0x7cc090c8
        WDFIOTARGET 0x7c9630b8

!wdfobject 0x83584e38
```

In the preceding example, the input handle refers to a WDFDEVICE object. This particular device object has five child objects--two WDFCMRESLIST objects, two WDFCHILDLIST objects, and one WDFIOTARGET object.

!wdfkd.wdfhelp

12/5/2018 • 2 minutes to read • Edit Online

The **!wdfkd.wdfhelp** extension displays help information about all Wdfkd.dll extension commands.

```
!wdfkd.wdfhelp
```

DLL

Wdfkd.dll

Additional Information

For more information, see Kernel-Mode Driver Framework Debugging.

Remarks

The **!wdfkd.wdfhelp** extension is equivalent to the **!wdfkd.help** extension.

!wdfkd.wdfinterrupt

12/5/2018 • 2 minutes to read • Edit Online

The **!wdfkd.wdfinterrupt** extension displays information about a WDFINTERRUPT object.

```
!wdfkd.wdfinterrupt Handle [Flags]
```

Parameters

Handle
A handle to a WDFINTERRUPT object.

Flags
Optional. Specifies the kind of information to display. *Flags* can be any combination of the following bits. The default value is 0x0.

Bit 0 (0x1)
Displays the interrupt service routines (ISRs) for the interrupt dispatch table (IDT) associated with this WDFINTERRUPT object. Setting this flag is equivalent to following the **!wdfinterrupt** extension with the **!idt** extension.

DLL

Wdfkd.dll

Frameworks

KMDF 1, UMDF 2

Additional Information

For more information, see Kernel-Mode Driver Framework Debugging.

Remarks

The following example shows the output of the **!wdfinterrupt** extension with bit 0 set in the *Flags* parameter (so the output displays information about the IDT).

```
kd> !wdfkd.wdfinterrupt 0x7a988698  1

# Dumping WDFINTERRUPT 0x7a988698
=========================
  Interrupt Type: Line-based, Connected, Enabled
  Vector: 0xa1 (!idt 0xa1)
  Irql: 0x9
  Mode: LevelSensitive
  Polarity: WdfInterruptPolarityUnknown
  ShareDisposition: CmResourceShareShared
  FloatingSave: FALSE
  Interrupt Priority Policy: WdfIrqPriorityUndefined
  Processor Affinity Policy: WdfIrqPolicyOneCloseProcessor
  Processor Group: 0
  Processor Affinity: 0x3

  dt nt!KINTERRUPT 0x8594eb28

  EvtInterruptIsr: 1394ohci!Interrupt::WdfEvtInterruptIsr (0x8d580552)
  EvtInterruptDpc: 1394ohci!Interrupt::WdfEvtInterruptDpc (0x8d580682)

Dumping IDT:

a1:          85167a58 ndis!ndisMiniportIsr (KINTERRUPT 85167a00)
                                  Wdf01000!FxInterrupt::_InterruptThunk (KINTERRUPT 85987500)

To get ISR from KINTERRUPT:
   dt <KINTERRUPT> nt!KINTERRUPT ServiceContext
   dt <ServiceContext> wdf01000!FxInterrupt m_EvtInterruptIsr
```

In the preceding example, the display concludes with two suggested **dt (Display Type)** commands that can be used to display additional data.

!wdfkd.wdfiotarget

12/5/2018 • 2 minutes to read • Edit Online

The **!wdfkd.wdfiotarget** extension displays information about a specified I/O target object.

```
!wdfkd.wdfiotarget Handle [Flags]
```

Parameters

Handle
A handle to an I/O target object.

Flags
Optional. Flags that specify the kind of information to display. *Flags* can be any combination of the following bits. The default value is 0x0.

Bit 0 (0x1)
The display will include details for each of the I/O target's pending request objects.

DLL

Wdfkd.dll

Frameworks

KMDF 1, UMDF 2

Additional Information

For more information, see Kernel-Mode Driver Framework Debugging.

Remarks

The following example shows a display from the **!wdfkd.wdfiotarget** extension.

```
kd> !wdfiotarget 0x7c9630b8

# WDFIOTARGET 8369cf40
=========================
WDFDEVICE: 0x7ca7b1c0
Target Device:  !devobj 0x81ede5d8
Target PDO: !devobj 0x822b8a90

Type: Instack target
State:  WdfIoTargetStarted

Requests waiting: 0

Requests sent: 0

Requests sent with ignore-target-state: 0
```

The output in the preceding example includes the address of the I/O target's parent framework device object, along with the addresses of the WDM DEVICE_OBJECT structures that represent the target driver's device object and the target device's physical device object (PDO).

!wdfkd.wdfldr

12/5/2018 • 2 minutes to read • Edit Online

The **!wdfkd.wdfldr** extension displays information about the drivers that are currently dynamically bound to the Windows Driver Frameworks. This includes both the Kernel-Mode Driver Framework (KMDF) and the User-Mode Driver Framework (UMDF).

```
!wdfkd.wdfldr [DriverName]
```

Parameters

DriverName
The name of a driver, including the filename extension. If you supply a driver name, this command displays detailed information about the one driver. If you do not supply a drive name, this command displays information about all drivers that are bound to the Windows Driver Frameworks.

DLL

Wdfkd.dll

Frameworks

KMDF 1, UMDF 1, UMDF 2

Additional Information

For more information, see Kernel-Mode Driver Framework Debugging.

Remarks

Here is an example of the output of **!wdfldr**.

```
## 0: kd> !wdfkd.wdfldr

##  KMDF Drivers

##  LoadedModuleList      0xfffff800003b61f8

LIBRARY_MODULE  0xffffe0000039f7c0
  Version       v1.13
  Service       \Registry\Machine\System\CurrentControlSet\Services\Wdf01000
  ImageName     Wdf01000.sys
  ImageAddress  0xfffff800002e7000
  ImageSize     0xc5000
  Associated Clients: 16

  ImageName                     Ver   WdfGlobals         FxGlobals          ImageAddress       ImageSize
  peauth.sys                    v1.7  0xffffe00003a95880 0xffffe00003a956e0 0xfffff80002678000 0x000ab000
  monitor.sys                   v1.11 0xffffe000001abc70 0xffffe000001abad0 0xfffff800022e7000 0x0000e000
  UsbHub3.sys                   v1.11 0xffffe000028a47b0 0xffffe000028a4610 0xfffff8000220b000 0x00077000
##   ...

## Total: 1 library loaded

##  UMDF Drivers

  DriverManagerProcess: 0xffffe00003470500

  ImageName                     Ver
  MyUmdfDriver.dll              v1.11
  SomeUmdf2Driver.dll           v2.0
  MyUmdf2Driver.dll             v2.0
```

Here is another example that supplies a driver name.

```
0: kd> !wdfldr MyUmdf2Driver.dll

Version    v2.0
Service    \Registry\Machine\System\CurrentControlSet\Services\MyUmdf2Driver

## !wdflogdump  MyUmdf2Driver.dll

##  UMDF Device Instances using MyUmdf2Driver.dll

Process             DevStack           DeviceId
0xffffe00000c32900  a5a3ab5f70         \Device\00000052 !wdfdriverinfo
```

!wdfkd.wdflogdump

6/25/2019 • 2 minutes to read • Edit Online

The **!wdfkd.wdflogdump** extension displays the WDF In-flight Recorder log records, if available, for a KMDF driver or a UMDF 2 driver. You can use this command with a complete memory dump, a kernel memory dump, or a live kernel-mode target.

KMDF

```
!wdfkd.wdflogdump [DriverName][WdfDriverGlobals][-d | -f | -a LogAddress]
```

UMDF

```
!wdfkd.wdflogdump  [DriverName.dll][HostProcessId][-d | -f | -m]
```

Parameters

DriverName

- KMDF: The name of a KMDF driver. The name must not include the .sys filename extension.
- UMDF: The name of a UMDF 2 driver. The name must include the .dll filename extension.

Parameter2

- KMDF: *WdfDriverGlobals* - The address of the *WdfDriverGlobals* structure. You can determine this address by running **!wdfkd.wdfldr** and looking for the field labeled "WdfGlobals". Or, you can supply @@(Driver!WdfDriverGlobals) as the address value, where *Driver* is the name of the driver. If any *WdfDriverGlobals* address is supplied, *DriverName* is ignored (although it must nevertheless be supplied).
- UMDF: *HostProcessId* - The process ID of an instance of wudfhost.exe. If you supply the process ID, this command displays the log records for that process. If you do not supply the process ID, this command displays a list of commands in this form:

  !wdflogdump *DriverName* **** *ProcessID*

  If a single process can be determined it will automatically be chosen.

Options
KMDF:

-d Displays only the driver logs.

-f Displays only the framework logs.

-a *LogAddress*Displays a specific driver log. If this option is used, the LogAddress must be provided.

UMDF:

-d Displays only the driver logs.

-f Displays only the framework logs.

-m Merges framework and driver logs in their recorded order.

DLL

Wdfkd.dll

Frameworks

KMDF 1, UMDF 2

Remarks

If you omit the *DriverName* parameter, the default driver name is used. Use the **!wdfkd.wdfgetdriver** extension to display the default driver name, and use the **!wdfkd.wdfsetdriver** extension to set the default driver name.

To display the framework's error log records from a small memory dump, use the **!wdfkd.wdfcrashdump** extension.

For information about setting information that the debugger needs to format WPP tracing messages, see **!wdfkd.wdftmffile** and **!wdfkd.wdfsettraceprefix**.

Additional Information

For information about enabling the inflight trace recorder for your driver, see Using Inflight Trace Recorder (IFR) in KMDF and UMDF 2 Drivers. For more information about debugging WDF drivers, see Debugging WDF Drivers. For information about KMDF debugging, see Kernel-Mode Driver Framework Debugging.

See also

!wdfkd.wdfcrashdump

!wdfkd.wdfsettraceprefix

!wdfkd.wdflogsave

12/5/2018 • 2 minutes to read • Edit Online

The **!wdfkd.wdflogsave** extension saves the Kernel-Mode Driver Framework (KMDF) error log records for a specified driver to an event trace log (.etl) file that you can view by using TraceView.

```
!wdfkd.wdflogsave [DriverName [FileName]]
```

Parameters

DriverName
Optional. The name of a driver. *DriverName* must not include the .sys file name extension.

FileName
Optional. The name of the file to which the KMDF error log records should be saved. *FileName* should not include the .etl file name extension. If you omit *FileName*, the KMDF error log records are saved to the DriverName.etl file.

DLL

Wdfkd.dll

Frameworks

KMDF 1, UMDF 2

Additional Information

For more information, see Kernel-Mode Driver Framework Debugging.

Remarks

If you omit the *DriverName* parameter, the default driver name is used. Use the **!wdfkd.wdfgetdriver** extension to display the default driver name, and use the **!wdfkd.wdfsetdriver** extension to set the default driver name.

!wdfkd.wdfmemory

12/5/2018 • 2 minutes to read • Edit Online

The **!wdfkd.wdfmemory** extension displays the address and size of the buffer that is associated with a framework memory object.

```
!wdfkd.wdfmemory Handle
```

Parameters

Handle
A handle to a framework memory object.

DLL

Wdfkd.dll

Frameworks

KMDF 1, UMDF 2

Additional Information

For more information, see Kernel-Mode Driver Framework Debugging.

!wdfkd.wdfobject

12/5/2018 • 2 minutes to read • Edit Online

The **!wdfkd.wdfobject** extension displays information about a specified framework object.

```
!wdfkd.wdfobject FrameworkObject
```

Parameters

FrameworkObject
A pointer to a framework object.

DLL

Wdfkd.dll

Frameworks

KMDF 1, UMDF 2

Additional Information

For more information, see Kernel-Mode Driver Framework Debugging.

Remarks

If the Kernel-Mode Driver Framework (KMDF) verifier is enabled for a driver and the public handle type was marked for tracking, the display from the **!wdfkd.wdfobject** extension includes the tag tracker (that is, the tracking object), as in the following example.

```
kd> !wdfobject 0x83584e38

The type for object 0x83584e38 is FxDevice
State: FxObjectStateCreated (0x1)

!wdfhandle 0x7ca7b1c0

dt FxDevice 0x83584e38

    context:  dt 0x83584ff8 ROOT_CONTEXT (size is 0x1 bytes)
     <no associated attribute callbacks>

Object debug extension 83584e20
   !wdftagtracker 0x83722d80
   Verifier lock 0x831cefa8

   State history:
    [0] FxObjectStateCreated (0x1)
```

!wdfkd.wdfopenhandles

12/5/2018 • 2 minutes to read • Edit Online

The **!wdfkd.wdfopenhandles** extension displays information about all the handles that are open on the specified WDF device.

```
!wdfkd.wdfopenhandles Handle [Flags]
```

Parameters

Handle
A handle to a framework device object (WDFDEVICE).

Flags
Optional. Specifies the kind of information to display. *Flags* can be any combination of the following bits. The default value is 0x0.

Bit 0 (0x1)
Displays file object context information.

DLL

Wdfkd.dll

Frameworks

KMDF 1, UMDF 2

Additional Information

For more information, see Kernel-Mode Driver Framework Debugging.

!wdfkd.wdfpool

12/5/2018 • 2 minutes to read • Edit Online

The **!wdfkd.wdfpool** extension is obsolete.

!wdfkd.wdfpooltracker

12/5/2018 • 2 minutes to read • Edit Online

The **!wdfkd.wdfpooltracker** extension is obsolete.

!wdfkd.wdfpoolusage

12/5/2018 • 2 minutes to read • Edit Online

The **!wdfkd.wdfpoolusage** extension displays pool usage information for a specified driver, if the Kernel-Mode Driver Framework (KMDF) verifier is enabled for the driver.

```
!wdfkd.wdfpoolusage [DriverName [SearchAddress] [Flags]]]
```

Parameters

DriverName
Optional. The name of a driver. *DriverName* must not include the .sys file name extension.

SearchAddress
Optional. A string that represents a memory address. The pool entry that contains *SearchAddress* is displayed. If *SearchAddress* is 0 or omitted, all of the driver's pool entries are displayed.

Flags
Optional. The kind of information to display. This parameter is valid only if *SearchAddress* is nonzero. *Flags* can be any combination of the following bits. The default value is 0x0.

Bit 0 (0x1)
Displays verbose output. Multiple lines are displayed for each. If this flag is not set, the information about an allocation is displayed on one line.

Bit 1 (0x2)
Displays internal type information for each handle.

Bit 2 (0x4)
Displays the caller of each pool entry.

DLL

Wdfkd.dll

Frameworks

KMDF 1, UMDF 2

Additional Information

For more information, see Kernel-Mode Driver Framework Debugging.

Remarks

If you omit the *DriverName* parameter, the default driver is used. You can display the default driver by using the **!wdfkd.wdfgetdriver** extension; you can set the default driver by using the **!wdfkd.wdfsetdriver** extension.

The following example shows the output from the **!wdfpoolusage** extension when no pool allocation is marked and the *Flags* value is set to 0.

```
## kd> !wdfpoolusage wdfrawbusenumtest 0 0
---------------------------------
## FxDriverGlobals 83b7af18 pool stats
---------------------------------
Driver Tag: 'RawB'
15126 NonPaged Bytes, 548 Paged Bytes
94 NonPaged Allocations, 10 Paged Allocations
15610 PeakNonPaged Bytes, 752 PeakPaged Bytes
100 PeakNonPaged Allocations, 14 PeakPaged Allocations

pool 82dbae00, Size  512 Tag 'RawB', NonPaged, Caller:  Wdf01000!FxVerifierLock::AllocateThreadTable+5d
```

The following example shows the output from **!wdfpoolusage** that appears when the value of *Flags* is 1. (Note that the ellipsis (...) on the second line indicates the omission of some output that is the same as that shown in the preceding example.)

```
kd> !wdfpoolusage wdfrawbusenumtest 0 1
. . .
100 PeakNonPaged Allocations, 14 PeakPaged Allocations

Client alloc starts at 82dbae00
Size  512 Tag 'RawB'
NonPaged (0x0)
Caller:  Wdf01000!FxVerifierLock::AllocateThreadTable+5d
```

!wdfkd.wdfqueue

12/5/2018 • 2 minutes to read • Edit Online

The **!wdfkd.wdfqueue** extension displays information about a specified framework queue object and the framework request objects that are in the queue.

```
!wdfkd.wdfqueue Handle
```

Parameters

Handle
A handle to a framework queue object.

DLL

Wdfkd.dll

Frameworks

KMDF 1, UMDF 2

Additional Information

For more information, see Kernel-Mode Driver Framework Debugging.

Remarks

The following example shows the display from a **!wdfkd.wdfqueue** extension.

```
kd> !wdfqueue 0x7ce7d1e8

# Dumping WDFQUEUE 0x7ce7d1e8
=========================
Parallel, Power-managed, PowerOff, Can accept, Can dispatch, ExecutionLevelDispatch, SynchronizationScopeNone
    Number of driver owned requests: 0
    Number of waiting requests: 0


    EvtIoDefault: (0xf221fad0) wdfrawbusenumtest!EvtIoQueueDefault
```

The queue in the preceding example is configured for parallel dispatching, is power-managed but is currently in the Off state, and can both accept and dispatch requests.

!wdfkd.wdfrequest

12/5/2018 • 2 minutes to read • Edit Online

The **!wdfkd.wdfrequest** extension displays information about a specified framework request object and the WDM I/O request packet (IRP) that is associated with the request object.

```
!wdfkd.wdfrequest Handle
```

Parameters

Handle
A handle to a framework request object.

DLL

Wdfkd.dll

Frameworks

KMDF 1, UMDF 2

Additional Information

For more information, see Kernel-Mode Driver Framework Debugging.

!wdfkd.wdfsearchpath

12/5/2018 • 2 minutes to read • Edit Online

The **!wdfkd.wdfsearchpath** extension sets the search path to formatting files for Kernel-Mode Driver Framework (KMDF) error log records.

```
!wdfkd.wdfsearchpath Path
```

Parameters

Path
The path of a directory that contains KMDF formatting files.

DLL

Wdfkd.dll

Frameworks

KMDF 1, UMDF 2

Additional Information

For more information, see Kernel-Mode Driver Framework Debugging.

Remarks

The KMDF formatting files are included in the Windows Driver Kit (WDK). The path to the formatting files depends on the installation directory of your WDK and on the version of the WDK that you have installed. The KMDF formatting files have extension tmf (trace message formatting). To determine the search path, browse or search your WDK installation for file names of the form Wdf*VersionNumber*.tmf. The following example shows how to use the **!wdfkd.wdfsearchpath** extension.

```
kd> !wdfsearchpath C:\WinDDK\7600\tools\tracing\amd64
```

The TRACE_FORMAT_SEARCH_PATH environment variable also controls the search path, but the **!wdfkd.wdfsearchpath** extension takes precedence over the search path that TRACE_FORMAT_SEARCH_PATH specifies.

Requirements

DLL	Wdfkd.dll

!wdfkd.wdfsettraceprefix

12/5/2018 • 2 minutes to read • Edit Online

The **!wdfkd.wdfsettraceprefix** extension sets the trace prefix format string.

```
!wdfkd.wdfsettraceprefix String
```

Parameters

String
A trace prefix string.

DLL

Wdfkd.dll

Frameworks

KMDF 1, UMDF 2

Additional Information

For more information, see Kernel-Mode Driver Framework Debugging.

Remarks

The trace prefix string is prepended to each trace message in the Kernel-Mode Driver Framework (KMDF) error log. The TRACE_FORMAT_PREFIX environment variable also controls the trace prefix string.

The format of the trace prefix string is defined by the Microsoft Windows tracing tools. For more information about the format of this string and how to customize it, see the "Trace Message Prefix" topic in the Driver Development Tools section of the Windows Driver Kit (WDK).

!wdfkd.wdfsetdriver

12/5/2018 • 2 minutes to read • Edit Online

The **!wdfkd.wdfsetdriver** extension sets the name of the default Kernel-Mode Driver Framework (KMDF) driver to which debugger extension commands apply.

```
!wdfkd.wdfsetdriver DriverName
```

Parameters

DriverName
The name of a driver. *DriverName* must not include the .sys file name extension.

DLL

Wdfkd.dll

Frameworks

KMDF 1, UMDF 2

Additional Information

For more information, see Kernel-Mode Driver Framework Debugging.

Remarks

The **!wdfkd.wdfsetdriver** extension sets the default driver name. You can use this name with other **wdfkd** extensions that would otherwise require you to specify a driver name.

To obtain the name of the current default KMDF driver, use the **!wdfkd.wdfgetdriver** extension.

!wdfkd.wdfspinlock

12/5/2018 • 2 minutes to read • Edit Online

The **!wdfkd.wdfspinlock** extension displays information about a framework spin-lock object. This information includes the spin lock's acquisition history and the length of time that the lock was held.

```
!wdfkd.wdfspinlock Handle
```

Parameters

Handle
A handle to a WDFSPINLOCK-typed object.

DLL

Wdfkd.dll

Frameworks

KMDF 1, UMDF 2

Additional Information

For more information, see Kernel-Mode Driver Framework Debugging.

!wdfkd.wdftagtracker

12/5/2018 • 2 minutes to read • Edit Online

The **!wdfkd.wdftagtracker** extension displays all available tag information (including tag value, line, file, and time) for a specified tag tracker.

```
!wdfkd.wdftagtracker TagObjectPointer [Flags]
```

Parameters

TagObjectPointer
A pointer to a tag tracker.

Flags
Optional. The kind of information to display. *Flags* can be any combination of the following bits. The default value is 0x0.

Bit 0 (0x1)
Displays the history of acquire operations and release operations on the object.

Bit 1 (0x2)
Displays the line number of the object in hexadecimal instead of decimal.

DLL

Wdfkd.dll

Frameworks

KMDF 1, UMDF 2

Additional Information

For more information, see Kernel-Mode Driver Framework Debugging.

Remarks

To retrieve a pointer to a tag tracker, use the **!wdfkd.wdfobject** extension on an internal framework object pointer.

To use tag tracking, you must enable both the Kernel-Mode Driver Framework (KMDF) verifier and handle tracking in the registry. Both of these settings are stored in the driver's **Parameters\Wdf** subkey of the **HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services** key.

To enable the KMDF verifier, set a nonzero value for **VerifierOn**.

To enable handle tracking, set the value of **TrackHandles** to the name of one or more object types, or specify an asterisk (*) to track all object types. For example, the following example specifies the tracking of references to all WDFDEVICE and WDFQUEUE objects.

```
TrackHandles: MULTI_SZ: WDFDEVICE WDFQUEUE
```

When you enable handle tracking for an object type, the framework tracks the references that are taken on any object of that type. This setting is useful in finding driver memory leaks that unreleased references cause. **TrackHandles** works only if the KMDF verifier is enabled.

!wdfkd.wdftmffile

6/25/2019 • 2 minutes to read • Edit Online

The **!wdfkd.wdftmffile** extension sets the trace message format (.tmf) file to use when the debugger is formatting Kernel-Mode Driver Framework (KMDF) error log records for the **!wdfkd.wdflogdump** or **!wdfkd.wdfcrashdump** extensions.

```
!wdfkd.wdftmffile TMFpath
```

Parameters

TMFpath
A path that contains the .tmf file.

DLL

Wdfkd.dll

Frameworks

KMDF 1, UMDF 2

Additional Information

For more information, see Kernel-Mode Driver Framework Debugging.

Remarks

If your driver uses a KMDF version earlier than 1.11, you must use the **!wdfkd.wdftmffile** extension before you can use the **!wdfkd.wdflogdump** or **!wdfkd.wdfcrashdump** extensions.

Starting in KMDF version 1.11, the framework library's symbol file (for example wdf01000.pdb) contains the trace message format (TMF) entries. Starting in the Windows 8 version of the kernel debugger, the Kernel-Mode Driver Framework Extensions (Wdfkd.dll) read the entries from the .pdb file. As a result, if your driver uses KMDF version 1.11 or later, and you are using the kernel debugger from Windows 8 or later, you do not need to use **!wdfkd.wdftmffile**. You do need to include the directory that contains the symbol file in the debugger's symbol path. The debugging target machine can be running any operating system that supports KMDF.

The following example shows how to use the **!wdfkd.wdftmffile** extension from the root WDK directory, for KMDF version 1.5.

```
kd> !wdftmffile tools\tracing\<platform>\wdf1005.tmf
```

Note that the path might be different for the version of the Windows Driver Kit (WDK) that you are using. Also note that the .tmf file's name represents the version of KMDF that you are using. For example, Wdf1005.tmf is the .tmf file for KMDF version 1.5.

For information about how to view the KMDF log during a debugging session, see Using the Framework's Event Logger.

!wdfkd.wdftraceprtdebug

12/5/2018 • 2 minutes to read • Edit Online

The **!wdfkd.wdftraceprtdebug** extension enables and disables the Traceprt.dll diagnostic mode, which generates verbose debugging information.

```
!wdfkd.wdftraceprtdebug {on | off}
```

Parameters

on
Enables the Traceprt.dll diagnostic mode.

off
Disables the Traceprt.dll diagnostic mode.

DLL

Wdfkd.dll

Frameworks

KMDF 1, UMDF 2

Additional Information

For more information, see Kernel-Mode Driver Framework Debugging.

Remarks

You should use the !wdfkd.wdftraceprtdebug extension only at the direction of technical support.

!wdfkd.wdfumdevstack

12/5/2018 • 2 minutes to read • Edit Online

The **!wdfkd.wdfumdevstack** extension displays detailed information about a UMDF device stack in the implicit process.

```
!wdfkd.wdfumdevstack DevstackAddress [Flags]
```

Parameters

DevstackAddress
Specifies the address of the device stack to display information about. You can use **!wdfkd.wdfumdevstacks** to get the addresses of UMDF device stacks in the implicit process.

Flags
Optional. Specifies the type of information to be displayed. *Flags* can be any combination of the following bits. The default value is 0x01.

Bit 0 (0x01)
Displays detailed information about the device stack.

Bit 7 (0x80)
Displays information about the internal framework.

DLL

Wdfkd.dll

Frameworks

UMDF 2

Additional Information

For more information, see Kernel-Mode Driver Framework Debugging.

Remarks

You can use this command in a kernel-mode debugging session or in a user-mode debugging session that is attached to the UMDF host process (wudfhost.exe).

This command displays the same information as the user-mode command **!wudfext.umdevstack**.

Here is an example of how to use **!wdfumdevstack**. First use **!wdfumdevstacks** to display the UMDF device stacks in the implicit process.

```
0: kd> !wdfkd.wdfumdevstacks
Number of device stacks: 1
  Device Stack: 0x000000a5a3ab5f70     Pdo Name: \Device\00000052
    Active: Yes
    Number of UM devices: 1
    Device 0
      Driver Config Registry Path: MyUmdf2Driver
      UMDriver Image Path: C:\WINDOWS\System32\drivers\UMDF\MyUmdf2Driver.dll
      FxDriver: 0xa5a3acaaa0
      FxDevice: 0xa5a3ac4fc0
      Open UM files (use !wdfumfile <addr> for details): <None>
      Device XFerMode: Deferred RW: Buffered CTL: Buffered
      DevStack XFerMode: Deferred RW: Buffered CTL: Buffered
```

The preceding output shows that there is one UMDF device stack in the implicit process. You can also see that the device stack has one device object (Number of UM devices: 1).

The preceding output displays the address of a device stack (0x000000a5a3ab5f70). To get detailed information about the device stack, pass its address to **!wdfumdevstack**. In this example, we set the *Flags* parameter to 0x80 to include information about the framework.

```
0: kd> !wdfkd.wdfumdevstack 0x000000a5a3ab5f70 0x80
  Device Stack: 0x000000a5a3ab5f70     Pdo Name: \Device\00000052
    Active: Yes
    Number of UM devices: 1
    Device 0
      Driver Config Registry Path: MyUmdf2Driver
      UMDriver Image Path: C:\WINDOWS\System32\drivers\UMDF\MyUmdf2Driver.dll
      FxDriver: 0xa5a3acaaa0
      FxDevice: 0xa5a3ac4fc0
      Open UM files (use !wdfumfile <addr> for details): <None>
      Device XFerMode: Deferred RW: Buffered CTL: Buffered
      Internal Values:
        wudfhost!WudfDriverAndFxInfo 0x000000a5a3ac21b8
        IUMDFramework: 0x0000000000000000
        IFxMessageDispatch: 0x000000a5a3aba630
        FxDevice 0x000000a5a3ac4fc0
        Modules:
          Driver: wudfhost!CWudfModuleInfo 0x000000a5a3ac18f0
          Fx:     wudfhost!CWudfModuleInfo 0x000000a5a3aca7a0
          wudfx02000!FxDriver: 0x000000a5a3acaaa0
      DevStack XFerMode: Deferred RW: Buffered CTL: Buffered
```

!wdfkd.wdfumdevstacks

12/5/2018 • 2 minutes to read • Edit Online

The **!wdfkd.wdfumdevstacks** extension displays information about all UMDF device stacks in the implicit process.

```
!wdfkd.wdfumdevstacks [Flags]
```

Parameters

Flags
Optional. Specifies the type of information to be displayed. *Flags* can be any combination of the following bits. The default value is 0x01.

Bit 0 (0x01)
Displays detailed information about each device stack.

Bit 7 (0x80)
Displays information about the internal framework.

DLL

Wdfkd.dll

Frameworks

UMDF 2

Additional Information

For more information, see Kernel-Mode Driver Framework Debugging.

Remarks

You can use this command in a kernel-mode debugging session or in a user-mode debugging session that is attached to the UMDF host process (wudfhost.exe).

This command displays the same information as the user-mode command **!wudfext.umdevstacks**.

Before you use this command, use **!process** to get a list of all UMDF host processes.

```
0: kd> !process 0 0 wudfhost.exe
PROCESS ffffe00000c32900
    SessionId: 0  Cid: 079c    Peb: 7ff782537000  ParentCid: 037c
    DirBase: 607af000  ObjectTable: ffffc00009807940  HandleCount: <Data Not Accessible>
    Image: WUDFHost.exe
```

The preceding output shows that there is one UMDF host process; that is, there is one instance of wudfhost.exe.

Next use **.process** to set the implicit process to wudfhost.exe.

```
0: kd> .process /P ffffe00000c32900
Implicit process is now ffffe000`00c32900
.cache forcedecodeptes done
```

Now use **!wdfkd.wdfumdevstacks** to display the UMDF device stacks in the implicit process (wudfhost.exe).

```
0: kd> !wdfkd.wdfumdevstacks
Number of device stacks: 1
  Device Stack: 0x000000a5a3ab5f70     Pdo Name: \Device\00000052
    Active: Yes
    Number of UM devices: 1
    Device 0
      Driver Config Registry Path: MyUmdf2Driver
      UMDriver Image Path: C:\WINDOWS\System32\drivers\UMDF\MyUmdf2Driver.dll
      FxDriver: 0xa5a3acaaa0
      FxDevice: 0xa5a3ac4fc0
      Open UM files (use !wdfumfile <addr> for details): <None>
      Device XFerMode: Deferred RW: Buffered CTL: Buffered
      DevStack XFerMode: Deferred RW: Buffered CTL: Buffered
```

The preceding output shows that there is one UMDF device stack in the implicit process. You can also see that the device stack has one device object (Number of UM devices: 1).

!wdfkd.wdfumdownirp

12/5/2018 • 2 minutes to read • Edit Online

The **!wdfkd.wdfumdownirp** extension displays the kernel-mode I/O request packet (IRP) that is associated with a specified user-mode IRP. This command is used in two steps. See Remarks.

```
!wdfkd.wdfumdownirp UmIrp [FileObject]
```

Parameters

UmIrp
Specifies the address of a user mode IRP. You can use **!wdfkd.wdfumirps** to get the addresses of UM IRPs in the implicit process.

FileObject
Specifies the address of a **_FILE_OBJECT** structure. For information about how to get this address, see Remarks.

DLL

Wdfkd.dll

Frameworks

UMDF 2

Additional Information

For more information, see Kernel-Mode Driver Framework Debugging.

Remarks

You can use this command in a kernel-mode debugging session or in a user-mode debugging session that is attached to the UMDF host process (wudfhost.exe).

To use this command, follow these steps:

1. Enter this command, passing only the address a user-mode IRP. The command displays a handle.
2. Pass the displayed handle to the **!handle** command. In the output of **!handle**, find the address of a **_FILE_OBJECT** structure.
3. Enter this command again, passing both the address of the user-mode IRP and the address of the **_FILE_OBJECT** structure.

!wdfkd.wdfumfile

12/5/2018 • 2 minutes to read • Edit Online

The **!wdfkd.wdfumfile** extension displays information about a UMDF intra-stack file.

```
!wdfkd.wdfumfile Address
```

Parameters

Address
Specifies the address of the UMDF intra-stack file to display information about.

DLL

Wdfkd.dll

Frameworks

UMDF 2

Additional Information

For more information, see Kernel-Mode Driver Framework Debugging.

Remarks

You can use this command in a kernel-mode debugging session or in a user-mode debugging session that is attached to the UMDF host process (wudfhost.exe).

This command displays the same information as the user-mode command **!wudfext.umfile**.

!wdfkd.wdfumirp

12/5/2018 • 2 minutes to read • Edit Online

The **!wdfkd.wdfumirp** extension displays information about a user-mode I/O request packet (UM IRP).

```
!wdfkd.wdfumirp Address
```

Parameters

Address
Specifies the address of the UM IRP to display information about. You can use **!wdfkd.wdfumirps** to get the addresses of UM IRPs in the implicit process.

DLL

Wdfkd.dll

Frameworks

UMDF 2

Additional Information

For more information, see Kernel-Mode Driver Framework Debugging.

Remarks

You can use this command in a kernel-mode debugging session or in a user-mode debugging session that is attached to the UMDF host process (wudfhost.exe).

This command displays the same information as the user-mode command **!wudfext.umirp**.

You can use **!process** to get a list of all UMDF host processes, and you can use **.process** to set the implicit process to one of the UMDF host processes. For a detailed example, see **!wdfkd.wdfumdevstacks**.

The following shows how to use **!wdfkd.wdfumirps** and **!wdfkd.wdfumirp** to display information about an individual UM IRP.

```
0: kd> !wdfkd.wdfumirps
Number of pending IRPS: 0x4
####  CWudfIrp     Current Type           UniqueId KernelIrp         Device Stack
----  ---------------  ------------------------------------------------  ----
...
0003  1ab9eae370   Power (WAIT_WAKE)          0     ffffe00000c53010  1ab9eaa6d0

0: kd> !wdfkd.wdfumirp 1ab9eae370
UM IRP: 0x0000001ab9eae370  UniqueId: 0x0  Kernel Irp: 0xffffe00000c53010
  Type: Power (WAIT_WAKE)
  ClientProcessId: 0x0
  Device Stack: 0x0000001ab9eaa6d0
  IoStatus
    hrStatus: 0x0
    Information: 0x0
  Total number of stack locations: 2
  CurrentStackLocation: StackLocation[ 0 ]
  > StackLocation[ 0 ]
      FxDevice:   (None)
      Completion:
        Callback:   0x0000000000000000
        Context:    0x0000001ab9ebc750
    StackLocation[ 1 ]
    ...
```

!wdfkd.wdfumirps

12/5/2018 • 2 minutes to read • Edit Online

The **!wdfkd.wdfumirps** extension displays the list of pending user-mode I/O request packets (UM IRPs) in the implicit process.

```
!wdfkd.wdfumirps NumberOfIrps Flags
```

Parameters

NumberOfIrps
Optional. Specifies the number of pending UM IRPs to display information about. If *NumberOfIrps* is an asterisk (*) or is omitted, all UM IRPs will be displayed.

Flags
Optional. Specifies the type of information to be displayed. *Flags* can be any combination of the following bits. The default value is 0x01.

Bit 0 (0x01)
Displays details about the pending IRPs.

DLL

Wdfkd.dll

Frameworks

UMDF 2

Additional Information

For more information, see Kernel-Mode Driver Framework Debugging.

Remarks

You can use this command in a kernel-mode debugging session or in a user-mode debugging session that is attached to the UMDF host process (wudfhost.exe).

This command displays the same information as the user-mode command **!wudfext.umirps**.

You can use **!process** to get a list of all UMDF host processes, and you can use **.process** to set the implicit process to one of the UMDF host processes. For a detailed example, see **!wdfkd.wdfumdevstacks**.

Here is an example of the output of **!wdfkd.wdfumirps**.

```
0: kd> !wdfkd.wdfumirps
Number of pending IRPS: 0x4
####  CWudfIrp     Current Type           UniqueId KernelIrp         Device Stack
----  ---------------  ---------------------------------------------------  ----
0000  1ab9e90c40   WdfRequestUndefined        0     0                 1ab9eaa6d0
0001  1ab9ebfa90   WdfRequestInternalIoctl    0     0                 1ab9eaa6d0
0002  1ab9ebfd10   WdfRequestInternalIoctl    0     0                 1ab9eaa6d0
0003  1ab9eae370   Power (WAIT_WAKE)          0     ffffe00000c53010  1ab9eaa6d0
```

!wdfkd.wdfumtriage

4/1/2019 • 2 minutes to read • Edit Online

The **!wdfkd.wdfumtriage** extension displays information about all UMDF devices on the system, including device objects, corresponding host process, loaded drivers and class extensions, PnP device stack, PnP device nodes, dispatched IRPs, and problem state if relevant.

```
!wdfkd.wdfumtriage
```

DLL

Wdfkd.dll

Frameworks

UMDF 2

Additional Information

For more information, see Kernel-Mode Driver Framework Debugging.

Remarks

You can use this command in a kernel-mode debugging session.

Here is an example of the output of **!wdfkd.wdfumtriage**.

```
0: kd> !wdfkd.wdfumtriage
-----------------------------
WudfRd Driver Object Info
-----------------------------
dt WudfRd!RdDriver 0xffffffff9a3e9ee8
Driver manager process 0x9a291680(Switch Context) (Dump Process)
HostFailKdDebugBreak 1

    -----------------------------
    UMDF FDO Info
    -----------------------------
    dt WudfRd!RdFdoDevice 0xffffffff9a3ce9c8
    dt WudfRd!RdProcess 0xffffffffa5863828
    Device instance path Root\umdf2\FusionV2
    !devnode 0xffffffff8aeca600
    !devstack 0xffffffff8aeca800
    # of UMDF drivers 2
    1)SensorsCx.dll
    2)FusionV2.dll

    -----------------------------
    UMDF FDO Info
    -----------------------------
    dt WudfRd!RdFdoDevice 0xffffffffa5a26758
    dt WudfRd!RdProcess 0xffffffffa5863828
    Device instance path ACPI\QCOM2495\2&daba3ff&0
    !devnode 0xffffffff8ee50e30
    !devstack 0xffffffff8ee43a00
    # of UMDF drivers 2
    1)SensorsCx.dll
    2)qcSensors8626.dll

    -----------------------------
    UMDF FDO Info
    -----------------------------
    dt WudfRd!RdFdoDevice 0xffffffffa6be4980
    dt WudfRd!RdProcess 0xffffffffa5863828
    Device instance path SWD\umdf2\SDOV2
    !devnode 0xffffffffa6be8980
    !devstack 0xffffffff9a297aa0
    # of UMDF drivers 2
    1)SensorsCx.dll
    2)SdoV2.dll

        -----------------------------
        Driver host process Info
        -----------------------------
        Host Process 0x9a3fdb80(Switch Context) (Dump Process)
        dt WudfRd!RdProcess 0xffffffffa5863828
        Process timeout 60 seconds

        I/O Port
        WudfPf!WdfLpcCommPort a583bc30
        # of pending message with timeout 0
        Pending Messages:
            WUDFPf!WudfLpcMessage 9a3ec750        !irp 0xffffffffa9a27008        (03/00)         - id 00000000000000ae

        I/O Cancel Port
        WudfPf!WdfLpcCommPort 9a3f1008
        # of pending message with timeout 0

        System Event Port
        WudfPf!WdfLpcCommPort a585af78
        # of pending message with timeout 0

        Non-state changing events port
        WudfPf!WdfLpcCommPort a585a498
        # of pending message with timeout 0
```

!wdfkd.wdfusbdevice

12/5/2018 • 2 minutes to read • Edit Online

The !wdfkd.wdfusbdevice extension displays information about a specified Kernel-Mode Driver Framework (KMDF) USB device object. This information includes all USB interfaces and the pipes that are configured for each interface.

```
!wdfkd.wdfusbdevice Handle [Flags]
```

Parameters

Handle
A handle to a WDFUSBDEVICE-typed USB device object.

Flags
Optional. A hexadecimal value that modifies the kind of information to return. The default value is 0x0. Flags can be any combination of the following bits:

Bit 0 (0x1)
The display will include the properties of the I/O target.

Bit 1 (0x2)
The display will include the properties of the I/O target for each USB pipe object.

DLL

Wdfkd.dll

Frameworks

KMDF 1, UMDF 2

Additional Information

For more information, see Kernel-Mode Driver Framework Debugging.

!wdfkd.wdfusbinterface

12/5/2018 • 2 minutes to read • Edit Online

The **!wdfkd.wdfusbinterface** extension displays information about a specified Kernel-Mode Driver Framework (KMDF) USB interface object, including its possible and current settings.

```
!wdfkd.wdfusbinterface Handle [Flags]
```

Parameters

Handle
A handle to a WDFUSBINTERFACE-typed USB interface object.

Flags
Optional. A hexadecimal value that modifies the kind of information to return. The default value is 0x0. Flags can be any combination of the following bits:

Bit 0 (0x1)
The display will include the properties of the I/O target for each KMDF USB pipe object.

DLL

Wdfkd.dll

Frameworks

KMDF 1, UMDF 2

Additional Information

For more information, see Kernel-Mode Driver Framework Debugging.

!wdfkd.wdfusbpipe

12/5/2018 • 2 minutes to read • Edit Online

The **!wdfkd.wdfusbpipe** extension displays information about a Kernel-Mode Driver Framework (KMDF) USB pipe object's I/O target.

```
!wdfkd.wdfusbpipe Handle [Flags]
```

Parameters

Handle
A handle to a WDFUSBPIPE-typed USB pipe object.

Flags
Optional. A hexadecimal value that modifies the kind of information to return. The default value is 0x0. Flags can be any combination of the following bits:

Bit 0 (0x1)
The display will include the properties of the I/O target.

DLL

Wdfkd.dll

Frameworks

KMDF 1, UMDF 2

Additional Information

For more information, see Kernel-Mode Driver Framework Debugging.

!wdfkd.wdfwmi

12/5/2018 • 2 minutes to read • Edit Online

The **!wdfkd.wdfwmi** extension displays the Microsoft Windows Management Instrumentation (WMI) information for a specified framework device object. This information includes all WMI provider objects and their associated WMI instance objects.

```
!wdfkd.wdfwmi Handle
```

Parameters

Handle
A handle to a framework device object.

DLL

Wdfkd.dll

Frameworks

KMDF 1

Additional Information

For more information, see Kernel-Mode Driver Framework Debugging.

Remarks

The output of the **!wdfkd.wdfwmi** extension includes information about the WMI registration, provider, and instances.

User-Mode Driver Framework Extensions (Wudfext.dll)

1/3/2019 • 2 minutes to read • Edit Online

Extension commands that are useful for debugging User-Mode Driver Framework drivers are implemented in Wudfext.dll.

Some extensions have additional restrictions on the Windows version or UMDF version that is required; these restrictions are noted on the individual reference pages.

Note When you create a new KMDF or UMDF driver, you must select a driver name that has 32 characters or less. This length limit is defined in wdfglobals.h. If your driver name exceeds the maximum length, your driver will fail to load.

For ways to use these extensions, see User-Mode Driver Framework Debugging.

!wudfext.help

12/5/2018 • 2 minutes to read • Edit Online

The **!wudfext.help** extension displays all Wudfext.dll extension commands.

```
!wudfext.help
```

DLL

Windows 2000	Unavailable
Windows XP and later	Wudfext.dll

Additional Information

For more information, see User-Mode Driver Framework Debugging.

!wudfext.umdevstack

12/5/2018 • 2 minutes to read • Edit Online

The **!wudfext.umdevstack** extension displays detailed information about a device stack in the host process.

```
!wudfext.umdevstack DevstackAddress [Flags]
```

Parameters

DevstackAddress
Specifies the address of the device stack to display information about.

Flags
Optional. Specifies the type of information to be displayed. *Flags* can be any combination of the following bits. The default value is 0x01.

Bit 0 (0x01)
Displays detailed information about the device stack.

Bit 8 (0x80)
Displays information about the internal framework.

DLL

Windows 2000	Unavailable
Windows XP and later	Wudfext.dll

Additional Information

For more information, see User-Mode Driver Framework Debugging.

Remarks

The following is an example of the **!wudfext.umdevstack** display:

```
kd> !umdevstack 0x0034e4e0
Device Stack: 0x0034e4e0  Pdo Name: \Device\00000057
 Number of UM drivers: 0x1
  Driver 00:
    Driver Config Registry Path: WUDFEchoDriver
    UMDriver Image Path: C:\Windows\system32\DRIVERS\UMDF\WUDFEchoDriver.dll
    Fx Driver: IWDFDriver 0xf2db8
    Fx Device: IWDFDevice 0xf2f80
        IDriverEntry: WUDFEchoDriver!CMyDriver 0x000f2c70
```

!wudfext.umdevstacks

12/5/2018 • 2 minutes to read • Edit Online

The **!wudfext.umdevstacks** extension displays information about all device stacks in the current host process.

```
!wudfext.umdevstacks [Flags]
```

Parameters

Flags
Optional. Specifies the type of information to be displayed. *Flags* can be any combination of the following bits. The default value is 0x01.

Bit 0 (0x01)
Displays detailed information about each device stack.

Bit 8 (0x80)
Displays information about the internal framework.

DLL

Wudfext.dll

Remarks

The **!wudfext.umdevstacks** extension displays the framework interface objects that are associated with each device stack. For more information about using the output from **!wudfext.umdevstacks**, see **!wudfext.umdevstack**.

The **!wudfext.umdevstacks** output includes two fields entitled "Object Tracking" and "Refcount Tracking". These indicate whether the object tracking option (**TrackObjects**) and the reference count tracking option (**TrackRefCounts**) have been enabled in WDF Verifier, respectively. If the object tracking option has been enabled, the display includes the object tracker address; this address can be passed to **!wudfext.wudfdumpobjects** to display tracking information.

Here is an example of the **!wudfext.umdevstacks** display:

```
0: kd> !umdevstacks
Number of device stacks: 1
  Device Stack: 0x038c6f08    Pdo Name: \Device\USBPDO-11
    Number of UM devices: 1
    Device 0
      Driver Config Registry Path: WUDFOsrUsbFx2
      UMDriver Image Path: D:\Windows\system32\DRIVERS\UMDF\WUDFOsrUsbFx2.dll
      Fx Driver: IWDFDriver 0x3076ff0
      Fx Device: IWDFDevice 0x3082e70
        IDriverEntry: WUDFOsrUsbFx2!CMyDriver 0x0306eff8
      Open UM files (use !umfile <addr> for details):
        0x04a8ef84
      Device XFerMode: CopyImmediately RW: Buffered CTL: Buffered
      Object Tracker Address: 0x03074fd8
        Object   Tracking ON
        Refcount Tracking OFF
    DevStack XFerMode: CopyImmediately RW: Buffered CTL: Buffered
```

Additional Information

For more information, see User-Mode Driver Framework Debugging.

!wudfext.umfile

12/5/2018 • 2 minutes to read • Edit Online

The **!wudfext.umfile** extension displays information about a UMDF intra-stack file.

```
!wudfext.umfile Address
```

Parameters

Address

Specifies the address of the UMDF intra-stack file to display information about.

DLL

Windows 2000	Unavailable
Windows XP with UMDF version 1.7 and later	Wudfext.dll

Additional Information

For more information, see User-Mode Driver Framework Debugging.

!wudfext.umirp

12/5/2018 • 2 minutes to read • Edit Online

The **!wudfext.umirp** extension displays information about a host user-mode I/O request packet (UM IRP).

```
!wudfext.umirp Address
```

Parameters

Address
Specifies the address of the UM IRP to display information about.

DLL

Windows 2000	Unavailable
Windows XP and later	Wudfext.dll

Additional Information

For more information, see User-Mode Driver Framework Debugging.

Remarks

You can use the **!wudfext.umirps** extension command to display a list of all outstanding UM IRPs in the host process.

Each UM IRP has one or more stack locations. Each stack location corresponds to the parameters that a single driver in the device stack will receive when it is called to handle a request.

!wudfext.umirp dumps all of the stack locations and marks the current location with a right angle bracket (>). The current location corresponds to the driver that currently owns the request. The current location changes when a driver forwards a request to the next lower driver in the stack, or when the driver completes a request that the driver owns.

The following is an example of the **!wudfext.umirp** display:

```
kd> !umirp 3dd480
UM IRP: 0x003dd480  UniqueId: 0xde  Kernel Irp: 0x0x85377850
  Type: WudfMsg_READ
  ClientProcessId: 0x338
  Device Stack: 0x0034e4e0
  IoStatus
    hrStatus: 0x0
    Information: 0x0
  Driver/Framework created IRP: No
  Data Buffer: 0x00000000 / 0
  IsFrom32BitProcess: Yes
  CancelFlagSet: No
  Cancel callback: 0x01102224
  Total number of stack locations: 2
  CurrentStackLocation: 2 (StackLocation[ 1 ])
    StackLocation[ 0 ]
      UNINITIALIZED
  > StackLocation[ 1 ]
      IWDFRequest:  ????
      IWDFDevice:   0x000f2f80
      IWDFFile:     0x003a7648
      Completion:
        Callback:   0x00000000
        Context:    0x00000000
      Parameters: (RequestType: WdfRequestRead)
        Buffer length:        0x400
        Key:                  0x00000000
        Offset:               0x0
```

!wudfext.umirps

12/5/2018 • 2 minutes to read • Edit Online

The **!wudfext.umirps** extension displays the list of pending user-mode I/O request packets (UM IRPs) in the host process.

```
!wudfext.umirps NumberOfIrps Flags
```

Parameters

NumberOfIrps
Optional. Specifies the number of pending UM IRPs to display information about. If *NumberOfIrps* is an asterisk (*) or is omitted, all UM all UM IRPs will be displayed.

Flags
Optional. Specifies the type of information to be displayed. *Flags* can be any combination of the following bits. The default value is 0x01.

Bit 0 (0x01)
Displays details about the pending IRPs.

DLL

Windows 2000	Unavailable
Windows XP and later	Wudfext.dll

Additional Information

For more information, see User-Mode Driver Framework Debugging.

Remarks

The list of pending UM IRPs that are displayed have either been presented to the driver or are waiting to be presented to the driver.

By default, **!wudfext.umirps** shows all UM IRPs. However, you can use the *NumberOfIrps* parameter to limit this display.

The following is an example of the **!wudfext.umirps** display:

```
kd> !umirps 0xa
Number of pending IRPS: 0xc8
####  CWudfIrp          Type        UniqueId          KernelIrp
----  ----------------  ----------  ----------------  ---------
0000            3dd280       READ                dc  856f02f0
0001            3dd380      WRITE                dd  85b869e0
0002            3dd480       READ                de  85377850
0003            3dd580       READ                df  93bba4e8
0004            3dd680      WRITE                e0  84cb9d70
0005            3dd780       READ                e1  85bec150
0006            3dd880      WRITE                e2  86651db0
0007            3dd980       READ                e3  85c22818
0008            3dda80       READ                e4  9961d150
0009            3ddb80      WRITE                e5  85c15148
```

To determine the corresponding kernel-mode IRP, use the **!wudfext.wudfdownkmirp** extension. Alternatively, you can use the values in the **UniqueId** and **KernelIrp** columns to match a UMDF IRP (or UM IRP) to a corresponding kernel IRP. You can pass the values in the **CWudfIrp** column to the **!wudfext.umirp** extension to determine the framework **IWDFRequest** objects that each layer in the device stack can access.

!wudfext.wudfdevice

12/5/2018 • 2 minutes to read • Edit Online

The **!wudfext.wudfdevice** extension displays the Plug and Play (PnP) and power-management state systems for a device.

```
!wudfext.wudfdevice pWDFDevice
```

Parameters

pWDFDevice
Specifies the address of the **IWDFDevice** interface to display PnP or power-management state about.

DLL

Windows 2000	Unavailable
Windows XP and later	Wudfext.dll

Additional Information

For more information, see User-Mode Driver Framework Debugging.

Remarks

You can use the **!wudfext.wudfdevice** extension to determine the current PnP or power-management state of the device that the *pWDFDevice* parameter specifies.

The following is an example of the **!wudfext.wudfdevice** display:

```
kd> !wudfdevice 0xf2f80
Pnp Driver Callbacks:
  IPnpCallback: 0x0
  IPnpCallbackHardware: 0x0
  IPnpSelfManagedIo: 0x0
Pnp State Machine:
  CurrentState:  WdfDevStatePnpStarted
  Pending UMIrp: 0x00000000
    Could not read event queue depth, assuming 8
  Event queue:
    Processed/in-process events:
      PnpEventAddDevice
      PnpEventStartDevice
      PnpEventPwrPolStarted
    Pending events:
    State History:
      WdfDevStatePnpInit
      WdfDevStatePnpInitStarting
      WdfDevStatePnpHardwareAvailable
      WdfDevStatePnpEnableInterfaces
      WdfDevStatePnpStarted
Power State Machine:
  CurrentState:      WdfDevStatePowerD0
  Pending UMIrp:     0x00000000
  IoCallbackFailure: false
    Could not read event queue depth, assuming 8
  Event queue:
    Processed/in-process events:
      PowerImplicitD0
    Pending events:
    State History:
      WdfDevStatePowerStartingCheckDeviceType
      WdfDevStatePowerD0Starting
      WdfDevStatePowerD0StartingConnectInterrupt
      WdfDevStatePowerD0StartingDmaEnable
      WdfDevStatePowerD0StartingStartSelfManagedIo
      WdfDevStatePowerDecideD0State
      WdfDevStatePowerD0
Power Policy State Machine:
  CurrentState             : WdfDevStatePwrPolStartingSucceeded
  PowerPolicyOwner         : false
  PendingSystemPower UMIrp : 0x00000000
  PowerFailed              : false
    Could not read event queue depth, assuming 8
  Event queue:
    Processed/in-process events:
      PwrPolStart
      PwrPolPowerUp
    Pending events:
    State History:
      WdfDevStatePwrPolStarting
      WdfDevStatePwrPolStarted
      WdfDevStatePwrPolStartingSucceeded
```

!wudfext.wudfdevicequeues

12/5/2018 • 2 minutes to read • Edit Online

The **!wudfext.wudfdevicequeues** extension displays information about all the I/O queues for a device.

```
!wudfext.wudfdevicequeues pWDFDevice
```

Parameters

pWDFDevice
Specifies the address of the **IWDFDevice** interface for which to display information about all of its associated I/O queues. The **!wudfext.wudfdriverinfo** extension command determines the address of **IWDFDevice**.

DLL

Windows 2000	Unavailable
Windows XP and later	Wudfext.dll

Additional Information

For more information, see User-Mode Driver Framework Debugging.

Remarks

The following is an example of the **!wudfext.wudfdevicequeues** display:

```
## kd> !wudfdevicequeues 0xf2f80
-------------------------------------------------
Queue: 1 (!wudfqueue 0x000f3500)
    WdfIoQueueDispatchSequential, POWER MANAGED, WdfIoQueuePowerOn, CAN ACCEPT, CAN DISPATCH
    Number of driver owned requests: 1
      IWDFIoRequest 0x000fa7c0     CWdfIoRequest 0x000fa748
    Number of waiting requests: 199
      IWDFIoRequest 0x000fa908     CWdfIoRequest 0x000fa890
      IWDFIoRequest 0x000faa50     CWdfIoRequest 0x000fa9d8
...
      IWDFIoRequest 0x000fa678     CWdfIoRequest 0x000fa600
    Driver's callback interfaces.
      IQueueCallbackRead 0x000f343c
      IQueueCallbackDeviceIoControl 0x000f3438
      IQueueCallbackWrite 0x000f3440
```

!wudfext.wudfdownkmirp

12/5/2018 • 2 minutes to read • Edit Online

The **!wudfext.downkmmirp** extension displays the kernel-mode I/O request packet (IRP) that corresponds to the specified user-mode I/O request packet (UM IRP).

```
!wudfext.wudfdownkmirp Address
```

Parameters

Address
Specifies the address of the UM IRP whose corresponding kernel-mode IRP is to be displayed.

DLL

Windows 2000	Unavailable
Windows XP and later	Wudfext.dll

Additional Information

For more information, see User-Mode Driver Framework Debugging.

Remarks

You can use the **!wudfext.umirps** extension command to display a list of all outstanding UM IRPs in the host process.

!wudfext.wudfdriverinfo

12/5/2018 • 2 minutes to read • Edit Online

The **!wudfext.wudfdriverinfo** extension displays information about a UMDF driver within the current host process.

```
!wudfext.wudfdriverinfo Name
```

Parameters

Name
Specifies the name of the UMDF driver to display information about.

DLL

Windows 2000	Unavailable
Windows XP and later	Wudfext.dll

Additional Information

For more information, see User-Mode Driver Framework Debugging.

Remarks

The **!wudfext.wudfdriverinfo** extension iterates through each level in each device stack and displays the driver and device information for each entry that matches the driver whose name is specified in the *Name* parameter.

You can use **!wudfext.wudfdriverinfo** to quickly find the device object for your driver.

The following is an example of the **!wudfext.wudfdriverinfo** display:

```
kd> !wudfdriverinfo wudfechodriver
IWDFDriver: 0xf2db8
  !WDFDEVICE 0xf2f80
    !devstack 0x34e4e0 @ level 0
```

!wudfext.wudfdumpobjects

12/5/2018 • 2 minutes to read • Edit Online

The **!wudfext.wudfdumpobjects** extension displays outstanding UMDF objects.

```
!wudfext.wudfdumpobjects ObjTrackerAddress
```

Parameters

ObjTrackerAddress
Specifies the address to track leaked objects. This address is displayed in the driver-stop message in the debugger when a leak occurs.

DLL

Windows 2000	Unavailable
Windows XP with UMDF version 1.7 and later	Wudfext.dll

Additional Information

For more information, see User-Mode Driver Framework Debugging.

Remarks

If the UMDF object tracking option (**TrackObjects**) has been enabled in WDF Verifier, you can use **!wudfext.wudfdumpobjects** to see any leaked objects that remain after the driver unloads.

If the **TrackObjects** option has been enabled, the address of the object tracker is automatically displayed when a leak is detected. Use this address as *ObjTrackerAddress* when executing **!wudfext.wudfdumpobjects**.

This extension can be used at any time, even if UMDF has not broken in to the debugger.

If UMDF is version 1.9 or above, you can use either **!wudfext.umdevstack** or **!wudfext.umdevstacks** to determine the address of the object tracker. This address can then be passed to **!wudfext.wudfdumpobjects**. Here is an example:

```
0: kd> !umdevstacks
Number of device stacks: 1
  Device Stack: 0x038c6f08    Pdo Name: \Device\USBPDO-11
    Number of UM devices: 1
    Device 0
      Driver Config Registry Path: WUDFOsrUsbFx2
      UMDriver Image Path: D:\Windows\system32\DRIVERS\UMDF\WUDFOsrUsbFx2.dll
      Fx Driver: IWDFDriver 0x3076ff0
      Fx Device: IWDFDevice 0x3082e70
        IDriverEntry: WUDFOsrUsbFx2!CMyDriver 0x0306eff8
      Open UM files (use !umfile <addr> for details):
        0x04a8ef84
      Device XFerMode: CopyImmediately RW: Buffered CTL: Buffered
      Object Tracker Address: 0x03074fd8
        Object   Tracking ON
        Refcount Tracking OFF
    DevStack XFerMode: CopyImmediately RW: Buffered CTL: Buffered

0: kd> !wudfdumpobjects 0x03074fd8
WdfTypeDriver    Object: 0x03076fb0, Interface: 0x03076ff0
WdfTypeDevice   Object: 0x03082e30, Interface: 0x03082e70
WdfTypeIoTarget Object: 0x03088f50, Interface: 0x03088f90
WdfTypeIoQueue                Object: 0x0308ce58, Interface: 0x0308ce98
WdfTypeIoQueue                Object: 0x03090e58, Interface: 0x03090e98
WdfTypeIoQueue                Object: 0x03092e58, Interface: 0x03092e98
WdfTypeIoTarget Object: 0x03098f40, Interface: 0x03098f80
WdfTypeFile         Object: 0x0309cfa0, Interface: 0x0309cfe0
WdfTypeUsbInterface         Object: 0x030a0f98, Interface: 0x030a0fd8
WdfTypeRequest Object: 0x030a2ef8, Interface: 0x030a2f38
WdfTypeIoTarget Object: 0x030a6f30, Interface: 0x030a6f70
WdfTypeIoTarget Object: 0x030aaf30, Interface: 0x030aaf70
WdfTypeIoTarget Object: 0x030aef30, Interface: 0x030aef70
WdfTypeRequest Object: 0x030c6ef8, Interface: 0x030c6f38
WdfTypeRequest Object: 0x030ceef8, Interface: 0x030cef38
WdfTypeMemoryObject    Object: 0x030d6fb0, Interface: 0x030d6ff0
WdfTypeMemoryObject    Object: 0x030dcfb0, Interface: 0x030dcff0
WdfTypeFile         Object: 0x030e4fa8, Interface: 0x030e4fe8
WdfTypeFile         Object: 0x030e6fa8, Interface: 0x030e6fe8
WdfTypeFile         Object: 0x030e8fa8, Interface: 0x030e8fe8
WdfTypeRequest Object: 0x030eaef8, Interface: 0x030eaf38
WdfTypeMemoryObject    Object: 0x030ecfb0, Interface: 0x030ecff0
WdfTypeMemoryObject    Object: 0x030eefb0, Interface: 0x030eeff0
```

!wudfext.wudffile

12/5/2018 • 2 minutes to read • Edit Online

The **!wudfext.wudffile** extension displays information about a framework file.

```
!wudfext.wudffile pWDFFile [TypeName]
```

Parameters

pWDFFile
Specifies the address of the **IWDFFile** interface to display information about.

TypeName
Optional. Specifies the type of the interface (for example, **IWDFDevice**). If a value for *TypeName* is supplied, the extension uses the value as the type of the interface. If an asterisk (*) is supplied as *TypeName*, or if *TypeName* is omitted, the extension attempts to automatically determine the type of the supplied interface.

DLL

Windows 2000	Unavailable
Windows XP with UMDF version 1.7 and later	Wudfext.dll

Additional Information

For more information, see User-Mode Driver Framework Debugging.

!wudfext.wudffilehandletarget

12/5/2018 • 2 minutes to read • Edit Online

The **!wudfext.wudffilehandletarget** extension displays information about a file-handle-based I/O target.

```
!wudfext.wudffilehandletarget pWDFFileHandleTarget TypeName
```

Parameters

pWDFFileHandleTarget
Specifies the address of the **IWDFIoTarget** interface to display information about. The **!wudfext.wudfobject** extension command determines the address of **IWDFIoTarget**.

TypeName
Optional. Specifies the type of the interface (for example, **IWDFDevice**). If a value for *TypeName* is supplied, the extension uses the value as the type of the interface. If an asterisk (*) is supplied as *TypeName*, or if *TypeName* is omitted, the extension attempts to automatically determine the type of the supplied interface.

DLL

Windows 2000	Unavailable
Windows XP with UMDF version 1.7 and later	Wudfext.dll

Additional Information

For more information, see User-Mode Driver Framework Debugging.

!wudfext.wudfiotarget

12/5/2018 • 2 minutes to read • Edit Online

The **!wudfext.wudfiotarget** extension displays information about an I/O target including the target's state and list of sent requests.

```
!wudfext.wudfiotarget pWDFTarget TypeName
```

Parameters

pWDFTarget
Specifies the address of the **IWDFIoTarget** interface to display information about. The **!wudfext.wudfobject** extension command determines the address of **IWDFIoTarget**.

TypeName
Optional. Specifies the type of the interface (for example, **IWDFDevice**). If a value for *TypeName* is supplied, the extension uses the value as the type of the interface. If an asterisk (*) is supplied as *TypeName*, or if *TypeName* is omitted, the extension attempts to automatically determine the type of the supplied interface.

DLL

Windows 2000	Unavailable
Windows XP with UMDF version 1.7 and later	Wudfext.dll

Additional Information

For more information, see User-Mode Driver Framework Debugging.

!wudfext.wudfobject

12/5/2018 • 3 minutes to read • Edit Online

The **!wudfext.wudfobject** extension displays information about a WDF object, as well as its parent and child relationships.

```
!wudfext.wudfobject pWDFObject Flags TypeName
```

Parameters

pWDFObject
Specifies the address of the WDF interface to display information about.

Flags
Optional. Specifies the type of information to be displayed. *Flags* can be any combination of the following bits. The default value is 0x01.

Bit 0 (0x01)
Steps recursively through the object hierarchy to obtain the parent and child relationships, which are displayed.

Bit 1 (0x02)
Displays only summary information about the object.

Bit 8 (0x80)
Steps recursively through the object hierarchy, and displays details about the internal framework.

TypeName
Optional. Specifies the type of the interface (for example, **IWDFDevice**). If a value for *TypeName* is supplied, the extension uses the value as the type of the interface. If an asterisk (*) is supplied as *TypeName*, or if *TypeName* is omitted, the extension attempts to automatically determine the type of the supplied interface.

DLL

Windows 2000	Unavailable
Windows XP and later	Wudfext.dll

Additional Information

For more information, see User-Mode Driver Framework Debugging.

Remarks

You can use **!wudfext.wudfobject** to list, for example, the child objects of an **IWDFDevice** object, which generally include the device's queues.

You can also use **!wudfext.wudfobject** to find WDF objects that are associated with a particular device, to check the state of a WDF object (for example, whether the WDF object is in the process of deletion), or to determine the WDF object's current reference count.

The **!wudfext.wudfobject** extension also displays the callback functions and context objects that the driver

associated with each framework object and attempts to determine the framework object's type. This last feature might not work with certain compilers.

The following are some examples. In the first example, **!wudfext.umdevstacks** gives 0x03050E70 as the address of a device object, and this address is then passed to **!wudfext.wudfobject**. The 0x1 flag is included to display all the children of this object.

```
0: kd> !umdevstacks
Number of device stacks: 1
  Device Stack: 0x038f6f08    Pdo Name: \Device\USBPDO-11
    Number of UM devices: 1
    Device 0
      Driver Config Registry Path: WUDFOsrUsbFx2
      UMDriver Image Path: D:\Windows\system32\DRIVERS\UMDF\WUDFOsrUsbFx2.dll
      Fx Driver: IWDFDriver 0x3044ff0
      Fx Device: IWDFDevice 0x3050e70
        IDriverEntry: WUDFOsrUsbFx2!CMyDriver 0x0303eff8
      Open UM files (use !umfile <addr> for details):
        0x049baf84
      Device XFerMode: CopyImmediately RW: Buffered CTL: Buffered
      Object Tracker Address: 0x00000000
        Object   Tracking OFF
        Refcount Tracking OFF
    DevStack XFerMode: CopyImmediately RW: Buffered CTL: Buffered

0: kd> !wudfobject 0x3050e70 1
IWDFDevice 0x3050e70 Fx: 0x3050e30 [Ref 2]
  15 Children
    00: IWDFIoTarget 0x3056f90 Fx: 0x3056f50 [Ref 3]
        No Children
    01: <Internal>
    02: <Internal>
    03: <Internal>
    04: IWDFIoQueue 0x305ae98 Fx: 0x305ae58 [Ref 8]
        No Children
    05: IWDFIoQueue 0x305ee98 Fx: 0x305ee58 [Ref 2]
        No Children
    06: IWDFIoQueue 0x3060e98 Fx: 0x3060e58 [Ref 2]
        No Children
    07: IWDFIoTarget 0x3066f80 Fx: 0x3066f40 [Ref 2]
        1 Children
          00: IWDFUsbInterface 0x306efd8 Fx: 0x306ef98 [Ref 1]
              3 Children
                00: IWDFIoTarget 0x3074f70 Fx: 0x3074f30 [Ref 2]
                    2 Children
                      00: IWDFMemory 0x30a4ff0 Fx: 0x30a4fb0 [Ref 2]
                          No Children
                      01: IWDFMemory 0x30aaff0 Fx: 0x30aafb0 [Ref 2]
                          No Children
                01: IWDFIoTarget 0x3078f70 Fx: 0x3078f30 [Ref 1]
                    No Children
```

Here is an example of **!wudfext.wudfobject** displaying a file object:

```
kd> !wudfobject 0xf5060
IWDFFile 0xf5060 Fx: 0xf4fe8 [Ref 1]
  State: Created   Parent: 0xf2f80
  No Children
```

Here is an example of **!wudfext.wudfobject** displaying a driver object:

```
kd> !wudfobject 0xf2db8 0x01
IWDFDriver 0xf2db8 Fx: 0xf2d40 [Ref 2]
  Callback: (WUDFEchoDriver!CMyDriver, 0xf2c68)
  State: Created   Parent: 0
  1 Children:
    00: IWDFDevice 0xf2f80 Fx: 0xf2f08 [Ref 2]
        State: Created   Parent: 0xf2db8
        5 Children:
          00: IWDFIoTarget 0xf33c0 Fx: 0xf3348 [Ref 3]
              State: Created   Parent: 0xf2f80
              No Children
          01: IWDFIoQueue 0xf3500 Fx: 0xf3488 [Ref 3]
              State: Created   Parent: 0xf2f80
              No Children
          02: IWDFFile 0xf5060 Fx: 0xf4fe8 [Ref 1]
              State: Created   Parent: 0xf2f80
              No Children
          03: IWDFFile 0xf5100 Fx: 0xf5088 [Ref 101]
              State: Created   Parent: 0xf2f80
              No Children
          04: IWDFFile 0xf51a0 Fx: 0xf5128 [Ref 101]
              State: Created   Parent: 0xf2f80
              No Children
```

!wudfext.wudfqueue

12/5/2018 • 2 minutes to read • Edit Online

The **!wudfext.wudfqueue** extension displays information about an I/O queue.

```
!wudfext.wudfqueue pWDFQueue
```

Parameters

pWDFQueue
Specifies the address of the **IWDFIoQueue** interface to display information about. The **!wudfext.wudfdevicequeues** extension command determines the address of **IWDFIoQueue**.

DLL

Windows 2000	Unavailable
Windows XP and later	Wudfext.dll

Additional Information

For more information, see User-Mode Driver Framework Debugging.

Remarks

The following is an example of the **!wudfext.wudfqueue** display:

```
kd> !wudfqueue 0x000f3500
    WdfIoQueueDispatchSequential, POWER MANAGED, WdfIoQueuePowerOn, CAN ACCEPT, CAN DISPATCH
    Number of driver owned requests: 1
      IWDFIoRequest 0x000fa7c0     CWdfIoRequest 0x000fa748
    Number of waiting requests: 199
      IWDFIoRequest 0x000fa908     CWdfIoRequest 0x000fa890
      IWDFIoRequest 0x000faa50     CWdfIoRequest 0x000fa9d8
      IWDFIoRequest 0x000fab98     CWdfIoRequest 0x000fab20
      ...
      IWDFIoRequest 0x000fa530     CWdfIoRequest 0x000fa4b8
      IWDFIoRequest 0x000fa678     CWdfIoRequest 0x000fa600
    Driver's callback interfaces.
      IQueueCallbackRead 0x000f343c
      IQueueCallbackDeviceIoControl 0x000f3438
      IQueueCallbackWrite 0x000f3440
```

!wudfext.wudfrefhist

12/5/2018 • 2 minutes to read • Edit Online

The **!wudfext.wudfrefhist** extension displays the reference count stack history for a UMDF object.

```
!wudfext.wudfrefhist ObjectAddress
```

Parameters

ObjectAddress
Specifies the address of the UMDF object whose reference count stack history is to be displayed. Note that this is the address of the object itself, not of the interface.

DLL

Wudfext.dll

Additional Information

For more information, see User-Mode Driver Framework Debugging.

Remarks

The **!wudfext.wudfrefhist** command is not supported by UMDF 1.11.

The *ObjectAddress* parameter must be the address of the UMDF object, not the address of the interface (which is used by many other UMDF extension commands). To determine the address of the UMDF object, use the **!wudfext.wudfdumpobjects** command, which displays both the UMDF object address and the interface address. Alternatively, if you know the address of the interface, you can use it as the argument of the **!wudfext.wudfobject** command, which displays the object address (displayed after the symbol "Fx:").

If the reference count tracking option (**TrackRefCounts**) has been enabled in WDF Verifier, you can use **!wudfext.wudfrefhist** to display each call stack that increments or decrements an object's reference count. You can determine whether a call stack is causing a memory leak by examining its **AddRef** and **Release** calls for references that are being added and not released.

This command can be used at any time, even if UMDF has not broken in to the debugger.

If this command does not display the desired information, make sure that the relevant data is paged in and then try again.

Here is an example of using the **!wudfext.wudfrefhist** command:

```
0:007> !wudfext.umdevstacks
...
      UMDriver Image Path: C:\Windows\System32\drivers\UMDF\WUDFOsrUsbFilter.dll
      Object Tracker Address: 0x0000003792ee9fc0
        Object   Tracking ON
        Refcount Tracking ON

0:007> !wudfext.wudfdumpobjects 0x0000003792ee9fc0
...
WdfTypeIoQueue   Object: 0x0000003792f05ce0, Interface: 0x0000003792f05d58

## 0:007> !wudfext.wudfrefhist 0x0000003792f05ce0
-----------------------------------------------------------------------
## 2  (++)
-----------------------------------------------------------------------
WUDFx!UfxObject::TrackRefCounts+0xb2
WUDFx!CWdfIoQueue::AddRef+0x17adc
WUDFx!CWdfIoQueue::CreateAndInitialize+0xeb
WUDFx!CWdfDevice::CreateIoQueue+0x1e7
WUDFOsrUsbFilter!CMyQueue::Initialize+0x48
WUDFOsrUsbFilter!CMyDevice::Configure+0x7d
WUDFOsrUsbFilter!CMyDriver::OnDeviceAdd+0xca
WUDFx!CWdfDriver::OnAddDevice+0x486
WUDFx!CWUDF::AddDevice+0x43
WUDFHost!CWudfDeviceStack::LoadDrivers+0x320
WUDFHost!CLpcNotification::Message+0x1340
WUDFPlatform!WdfLpcPort::ProcessMessage+0x140
WUDFPlatform!WdfLpcCommPort::ProcessMessage+0x92
WUDFPlatform!WdfLpc::RetrieveMessage+0x20c
```

!wudfext.wudfrequest

12/5/2018 • 2 minutes to read • Edit Online

The **!wudfext.wudfrequest** extension displays information about an I/O request.

```
!wudfext.wudfrequest pWDFRequest
```

Parameters

pWDFRequest
Specifies the address of the **WDFIoRequest** interface to display information about. The **!wudfext.wudfqueue** extension command determines the address of **WDFIoRequest**.

DLL

Wudfext.dll

Additional Information

For more information, see User-Mode Driver Framework Debugging.

Remarks

The following is an example of the **!wudfext.wudfrequest** display:

```
kd> !wudfrequest 0x000fa530
CWdfIoRequest  0x000fa4b8
Type: WdfRequestRead
  IWDFIoQueue: 0x000f3500
Completed: No
Canceled: No
UM IRP: 0x00429108  UniqueId: 0xf4  Kernel Irp: 0x0x936ef160
  Type: WudfMsg_READ
  ClientProcessId: 0x1248
  Device Stack: 0x003be4e0
  IoStatus
    hrStatus: 0x0
    Information: 0x0
  Driver/Framework created IRP: No
  Data Buffer: 0x00000000 / 0
  IsFrom32BitProcess: Yes
  CancelFlagSet: No
  Cancel callback: 0x000fa534
  Total number of stack locations: 2
  CurrentStackLocation: 2 (StackLocation[ 1 ])
    StackLocation[ 0 ]
      UNINITIALIZED
  > StackLocation[ 1 ]
      IWDFRequest:  ????
      IWDFDevice:   0x000f2f80
      IWDFFile:     0x00418cf0
      Completion:
        Callback:   0x00000000
        Context:    0x00000000
      Parameters: (RequestType: WdfRequestRead)
        Buffer length:        0x400
        Key:                  0x00000000
        Offset:               0x0
```

!wudfext.wudfusbinterface

12/5/2018 • 2 minutes to read • Edit Online

The **!wudfext.wudfusbinterface** extension displays information about a USB interface object.

```
!wudfext.wudfusbinterface pWDFUSBInterface TypeName
```

Parameters

pWDFUSBInterface
Specifies the address of the **IWDFUsbInterface** interface to display information about. The **!wudfext.wudfobject** extension command determines the address of **IWDFUsbInterface**.

TypeName
Optional. Specifies the type of the interface (for example, **IWDFDevice**). If a value for *TypeName* is supplied, the extension uses the value as the type of the interface. If an asterisk (*) is supplied as *TypeName*, or if *TypeName* is omitted, the extension attempts to automatically determine the type of the supplied interface.

DLL

Windows 2000	Unavailable
Windows XP with UMDF version 1.7 and later	Wudfext.dll

Additional Information

For more information, see User-Mode Driver Framework Debugging.

!wudfext.wudfusbpipe

12/5/2018 • 2 minutes to read • Edit Online

The **!wudfext.wudfusbpipe** extension displays information about a USB pipe object.

```
!wudfext.wudfusbpipe pWDFUSBPipe TypeName
```

Parameters

pWDFUSBPipe
Specifies the address of the **IWDFUsbTargetPipe** interface to display information about. The **!wudfext.wudfobject** extension command determines the address of **IWDFUsbTargetPipe**.

TypeName
Optional. Specifies the type of the interface (for example, **IWDFDevice**). If a value for *TypeName* is supplied, the extension uses the value as the type of the interface. If an asterisk (*) is supplied as *TypeName*, or if *TypeName* is omitted, the extension attempts to automatically determine the type of the supplied interface.

DLL

Windows 2000	Unavailable
Windows XP with UMDF version 1.7 and later	Wudfext.dll

Additional Information

For more information, see User-Mode Driver Framework Debugging.

!wudfext.wudfusbtarget

12/5/2018 • 2 minutes to read • Edit Online

The **!wudfext.wudfusbtarget** extension displays information about a USB I/O target.

```
!wudfext.wudfusbtarget pWDFUSBTarget TypeName
```

Parameters

pWDFUSBTarget
Specifies the address of the **IWDFUsbTargetDevice** interface to display information about. The **!wudfext.wudfobject** extension command determines the address of **IWDFUsbTargetDevice**.

TypeName
Optional. Specifies the type of the interface (for example, **IWDFDevice**). If a value for *TypeName* is supplied, the extension uses the value as the type of the interface. If an asterisk (*) is supplied as *TypeName*, or if *TypeName* is omitted, the extension attempts to automatically determine the type of the supplied interface.

DLL

Windows 2000	Unavailable
Windows XP with UMDF version 1.7 and later	Wudfext.dll

Additional Information

For more information, see User-Mode Driver Framework Debugging.

WMI Tracing Extensions (Wmitrace.dll)

12/5/2018 • 2 minutes to read • Edit Online

The extension commands for software tracing sessions can be found in Wmitrace.dll, a library of functions designed to use Windows Management Instrumentation (WMI) for event tracing.

!wmitrace.disable

12/5/2018 • 2 minutes to read • Edit Online

The **!wmitrace.disable** extension disables a provider for the specified Event Tracing for Windows (ETW) trace session.

```
!wmitrace.disable { LoggerID | LoggerName } GUID
```

Parameters

LoggerID
Specifies the trace session. *LoggerID* is an ordinal number that the system assigns to each trace session on the computer.

LoggerName
Specifies the trace session. *LoggerName* is the text name that was specified when the trace session was started.

GUID
Specifies the GUID of the provider to be disabled.

DLL

This extension is exported by Wmitrace.dll.

This extension is available in Windows 7 and later versions of Windows.

Additional Information

For a conceptual overview of event tracing, see the Microsoft Windows SDK. For information about tracing tools, see the Windows Driver Kit (WDK).

Remarks

After using this extension, you must resume program execution (for example, by using the **g (Go)** command) in order for it to take effect. After a brief time, the target computer automatically breaks into the debugger again.

To enable a provider, use **!wmitrace.enable**.

!wmitrace.dumpmini

12/5/2018 • 2 minutes to read • Edit Online

The **!wmitrace.dumpmini** extension displays the system trace fragment, which is stored in a dump file.

```
!wmitrace.dumpmini
```

DLL

This extension is exported by Wmitrace.dll.

This extension is available in Windows Vista and later versions of Windows.

This extension is useful only when debugging a minidump file or a full dump file.

Additional Information

For a conceptual overview of event tracing, see the Microsoft Windows SDK. For information about tracing tools, see the Windows Driver Kit (WDK).

Remarks

The *system trace fragment* is a copy of the contents of the last buffer of the System Context Log. Under normal conditions, this is the trace session whose logger ID is 2.

!wmitrace.dumpminievent

12/5/2018 • 2 minutes to read • Edit Online

The **!wmitrace.dumpminievent** extension displays the system event log trace fragment, which is stored in a dump file.

```
!wmitrace.dumpminievent
```

DLL

This extension is exported by Wmitrace.dll.

This extension is available in Windows Vista Service Pack 1 (SP1) and later versions of Windows.

This extension is useful only when debugging a minidump file or a full dump file.

Additional Information

For a conceptual overview of event tracing, see the Microsoft Windows SDK. For information about tracing tools, see the Windows Driver Kit (WDK).

Remarks

The *system event log trace fragment* is a copy of the contents of the last buffer of the System Event Log. The **!wmitrace.dumpminievent** extension displays its contents in event log format.

!wmitrace.dynamicprint

12/5/2018 • 2 minutes to read • Edit Online

The **!wmitrace.dynamicprint** extension controls whether the debugger displays the trace messages generated by a session running in KD_FILTER_MODE.

```
!wmitrace.dynamicprint {0 | 1}
```

Parameters

0
Turns the trace message display off.

1
Turns the trace message display on.

DLL

This extension is exported by Wmitrace.dll.

This extension is available in Windows 2000 and later versions of Windows. If you want to use this extension with Windows 2000, you must first copy the Wmitrace.dll file from the winxp subdirectory of the Debugging Tools for Windows installation directory to the w2kfre subdirectory.

Additional Information

For a conceptual overview of event tracing, see the Microsoft Windows SDK. For help in starting a trace session, see "Tracelog" in the Windows Driver Kit (WDK).

Remarks

Before you use this extension, start a trace session, and specify that the trace messages should be sent to the debugger. For example, if you use **!wmitrace.start** to start the session, use the **-kd** parameter. If you use Tracelog to start the trace session, use its **-kd** parameter. Tracelog (tracelog.exe) is a trace controller included in the Windows Driver Kit.

Trace messages are held in a buffers on the target computer. Those buffers are flushed and sent to the debugger on the host computer at regular intervals. You can specify the flush timer interval by using the **-kd** parameter of the **!wmitrace.start** command or the **-kd** parameter of the Tracelog tool. Starting in Windows 8, you can specify the flush timer value in milliseconds by appending **ms** to the flush timer value.

By default, ETW maintains per-processor trace buffers on the target computer. When the trace buffers are flushed and sent to the debugger on the host computer, there is no mechanism for merging the buffers into a chronological sequence of events. So the events might be displayed out of order. Starting in Windows 7, you can solve this problem by setting the **-lowcapacity** parameter when you use the Tracelog tool to start a trace session.

Tracelog *MySession* **-kd -lowcapacity**

When you start a session with **-lowcapacity** set, all events go to a single buffer on the target computer, and the events are displayed in the correct order in the debugger on the host computer.

Also, before using this extension, use **!wmitrace.searchpath** or **!wmitrace.tmffile** to specify the trace message format files. The system uses the trace message format files to format the binary trace messages so that they can

be displayed as human-readable text.

See also

!wmitrace.start

!wmitrace.enable

12/5/2018 • 2 minutes to read • Edit Online

The **!wmitrace.enable** extension enables a provider for the specified Event Tracing for Windows (ETW) trace session.

```
!wmitrace.enable { LoggerID | LoggerName } GUID [-level Num] [-matchallkw Num] [-matchanykw Num] [-
enableproperty Num] [-flag Num]
```

Parameters

LoggerID
Specifies the trace session. *LoggerID* is an ordinal number that the system assigns to each trace session on the computer.

LoggerName
Specifies the trace session. *LoggerName* is the text name that was specified when the trace session was started.

GUID
Specifies the GUID of the provider to be enabled.

-level *Num*
Specifies the level. *Num* can be any integer.

-matchallkw *Num*
Specifies one or more keywords. If multiple keywords are specified, the provider will be enabled only if all keywords are matched. *Num* can be any integer.

-matchanykw *Num*
Specifies one or more keywords. If multiple keywords are specified, the provider will be enabled if at least one keyword is matched. *Num* can be any integer. The effects of this parameter overlap with the effects of the -flag parameter.

-enableproperty *Num*
Specifies the enable property. *Num* can be any integer.

-flag *Num*
Specifies one or more flags. *Num* can be any integer. The effects of this parameter overlap with the effects of the -matchanykw parameter.

DLL

This extension is exported by Wmitrace.dll.

This extension is available in Windows 7 and later versions of Windows.

Additional Information

For a conceptual overview of event tracing, see the Microsoft Windows SDK. For information about tracing tools, see the Windows Driver Kit (WDK).

Remarks

After using this extension, you must resume program execution (for example, by using the **g (Go)** command) in

order for it to take effect. After a brief time, the target computer automatically breaks into the debugger again.

To disable a provider, use **!wmitrace.disable**.

!wmitrace.eventlogdump

12/5/2018 • 2 minutes to read • Edit Online

The **!wmitrace.eventlogdump** extension displays the contents of the specified logger. The display is formatted like an event log.

```
!wmitrace.eventlogdump { LoggerID | LoggerName }
```

Parameters

LoggerID
Specifies the trace session. *LoggerID* is an ordinal number that the system assigns to each trace session on the computer.

LoggerName
Specifies the trace session. *LoggerName* is the text name that was specified when the trace session was started.

DLL

This extension is exported by Wmitrace.dll.

This extension is available in Windows 2000 and later versions of Windows. If you want to use this extension with Windows 2000, you must first copy the Wmitrace.dll file from the winxp subdirectory of the Debugging Tools for Windows installation directory to the w2kfre subdirectory.

Additional Information

For a conceptual overview of event tracing, see the Microsoft Windows SDK. For information about tracing tools, see the Windows Driver Kit (WDK).

Remarks

This extension is similar to the **!wmitrace.logdump** extension, except that the output of **!wmitrace.eventlogdump** is formatted in event log style, and the output of **!wmitrace.logdump** is formatted in Windows software trace preprocessor (WPP) style. You should choose the extension whose format is appropriate for the data you wish to display.

To find the logger ID of a trace session, use the **!wmitrace.strdump** extension. Alternatively, you can use the Tracelog command tracelog -l to list the trace sessions and their basic properties, including the logger ID.

!wmitrace.help

12/5/2018 • 2 minutes to read • Edit Online

The **!wmitrace.help** extension displays the extension commands in Wmitrace.dll.

```
!wmitrace.help
```

DLL

This extension is exported by Wmitrace.dll.

This extension is available in Windows 2000 and later versions of Windows. If you want to use this extension with Windows 2000, you must first copy the Wmitrace.dll file from the winxp subdirectory of the Debugging Tools for Windows installation directory to the w2kfre subdirectory.

Additional Information

For a conceptual overview of event tracing, see the Microsoft Windows SDK. For information about tracing tools, see the Windows Driver Kit (WDK).

!wmitrace.logdump

6/25/2019 • 2 minutes to read • Edit Online

The **!wmitrace.logdump** extension displays the contents of the trace buffers for a trace session. You can limit the display to trace messages from specified providers.

```
!wmitrace.logdump [-t Count] [{LoggerID|LoggerName} [GUIDFile]]
```

Parameters

-t *Count*
Limits the output to the most recent messages. *Count* specifies the number of messages to display.

LoggerID
Specifies the trace session. *LoggerID* is an ordinal number that the system assigns to each trace session on the computer. If no parameter is specified, the trace session with ID equal to 1 is used.

LoggerName
Specifies the trace session. *LoggerName* is the text name that was specified when the trace session was started.

GUIDFile
Displays only trace messages from providers specified in the *GUIDFile* file. *GUIDFile* represents the path (optional) and file name of a text file that contains the control GUIDs of one or more trace providers, such as a .guid or .ctl file.

DLL

This extension is exported by Wmitrace.dll.

This extension is available in Windows 2000 and later versions of Windows. If you want to use this extension with Windows 2000, you must first copy the Wmitrace.dll file from the winxp subdirectory of the Debugging Tools for Windows installation directory to the w2kfre subdirectory.

Additional Information

For a conceptual overview of event tracing, see the Microsoft Windows SDK. For information about Tracelog, see "Tracelog" in the Windows Driver Kit (WDK).

Remarks

During Windows software trace preprocessor (WPP) software tracing, trace session buffers are used to store trace messages until they are flushed to a log file or to a trace consumer for a real-time display. The **!wmitrace.logdump** extension displays the contents of the buffers that are in physical memory. The display appears in the Debugger Command window.

This extension is especially useful to recover the most recent traces when a crash occurs, and to display the traces stored in a crash dump file.

Before you use this extension, use **!wmitrace.searchpath** or **!wmitrace.tmffile** to specify the trace message format files. The system uses the trace message format files to format the binary trace messages in the buffers so that they can be displayed as human-readable text.

Note If your driver uses UMDF version 1.11 or later, you do not need to use **!wmitrace.searchpath** or

!wmitrace.tmffile.

When you use Tracelog to start a trace session with circular buffering (-buffering), use this extension to display the buffer contents.

To find the logger ID of a trace session, use the **!wmitrace.strdump** extension. Alternatively, you can use the Tracelog command tracelog -l to list the trace sessions and their basic properties, including the logger ID.

This extension is only useful during WPP software tracing, and earlier (legacy) methods of Event Tracing for Windows. Trace events that are produced by other manifested providers do not use trace message format (TMF) files, and therefore this extension does not display their contents.

This extension is similar to the **!wmitrace.eventlogdump** extension, except that the output of **!wmitrace.logdump** is formatted in WPP style, and the output of **!wmitrace.eventlogdump** is formatted in event log style. You should choose the extension whose format is appropriate for the data you want to display.

For information about how to view the UMDF trace log, see Using WPP Software Tracing in UMDF-based Drivers.

!wmitrace.logger

12/5/2018 • 2 minutes to read • Edit Online

The **!wmitrace.logger** extension displays data about the trace session, including the session configuration data. This extension does not display trace messages generated during the session.

```
!wmitrace.logger [ LoggerID | LoggerName ]
```

Parameters

LoggerID
Specifies the trace session. *LoggerID* is an ordinal number that the system assigns to each trace session on the computer. If no parameter is specified, the trace session with ID equal to 1 is used.

LoggerName
Specifies the trace session. *LoggerName* is the text name that was specified when the trace session was started.

DLL

This extension is exported by Wmitrace.dll.

This extension is available in Windows 2000 and later versions of Windows. If you want to use this extension with Windows 2000, you must first copy the Wmitrace.dll file from the winxp subdirectory of the Debugging Tools for Windows installation directory to the w2kfre subdirectory.

Additional Information

For a conceptual overview of event tracing, see the Microsoft Windows SDK.

Remarks

This extension is designed for performance logs and events, which cannot be formatted for human-readable display. To display the trace messages in a trace session buffer, along with header data, use **!wmitrace.logdump**.

To find the logger ID of a trace session, use the **!wmitrace.strdump** extension. Alternatively, you can use the Tracelog command tracelog -l to list the trace sessions and their basic properties, including the logger ID.

!wmitrace.logsave

12/5/2018 • 2 minutes to read • Edit Online

The **!wmitrace.logsave** extension writes the current contents of the trace buffers for a trace session to a file.

```
!wmitrace.logsave {LoggerID|LoggerName} Filename
```

Parameters

LoggerID
Specifies the trace session. *LoggerID* is an ordinal number that the system assigns to each trace session on the computer.

LoggerName
Specifies the trace session. *LoggerName* is the text name that was specified when the trace session was started.

Filename
Specifies a path (optional) and file name for the output file.

DLL

This extension is exported by Wmitrace.dll.

This extension is available in Windows 2000 and later versions of Windows. If you want to use this extension with Windows 2000, you must first copy the Wmitrace.dll file from the winxp subdirectory of the Debugging Tools for Windows installation directory to the w2kfre subdirectory.

Additional Information

For a conceptual overview of event tracing, see the Microsoft Windows SDK. For information about Tracelog, see "Tracelog" in the Windows Driver Kit (WDK).

Remarks

This extension displays only the traces that are in memory at the time. It does not display trace messages that have been flushed from the buffers and delivered to an event trace log file or to a trace consumer.

Trace session buffers store trace messages until they are flushed to a log file or to a trace consumer for a real-time display. This extension saves the contents of the buffers that are in physical memory to the specified file.

The output is written in binary format. Typically, these files use the .etl (event trace log) filename extension.

When you use Tracelog to start a trace session with circular buffering (-buffering), you can use this extension to save the current buffer contents.

To find the logger ID of a trace session, use the **!wmitrace.strdump** extension. Alternatively, you can use the Tracelog command tracelog -l to list the trace sessions and their basic properties, including the logger ID.

!wmitrace.searchpath

12/5/2018 • 2 minutes to read • Edit Online

The **!wmitrace.searchpath** extension specifies the location of the trace message format files for messages in the trace buffers.

```
!wmitrace.searchpath [+] TMFPath
!wmitrace.searchpath
```

Parameters

+
Causes *TMFPath* to be appended to the existing search path. If the plus (+) token is not used, *TMFPath* replaces the existing search path.

TMFPath
The path to the directory where the debugger should look for the trace message format files. Paths that contain spaces are not supported. If multiple paths are included, they should be separated by semicolons, and the entire string should be enclosed in quotation marks. If the path is in quotation marks, the backslash character must be preceded by an escape character (`"c:\\debuggers;c:\\debuggers2"`). When the **+** token is used, *TMFPath* will be appended to the existing path, with a semicolon automatically inserted between the existing path and the new path; however, if the **+** token is used, quotation marks cannot be used.

DLL

This extension is exported by Wmitrace.dll.

This extension is available in Windows 2000 and later versions of Windows. If you want to use this extension with Windows 2000, you must first copy the Wmitrace.dll file from the winxp subdirectory of the Debugging Tools for Windows installation directory to the w2kfre subdirectory.

Additional Information

For a conceptual overview of event tracing, see the Microsoft Windows SDK. For information about trace message format files, see the "Trace Message Format Files" topic in the Windows Driver Kit (WDK).

Remarks

When used with no parameters, **!wmitrace.searchpath** displays the current search path.

The trace message format files (*.tmf) contain instructions for formatting the binary trace messages that a trace provider generates.

The *TMFPath* parameter must contain only a path to a directory; it cannot include a file name. The name of a TMF file is a message GUID followed by the .tmf extension. When the system formats a message, it reads the message GUID on the message and searches recursively for a TMF file whose name matches the message GUID, beginning in the specified directory.

Windows needs a TMF file in order to format the binary trace messages in a buffer. Use **!wmitrace.searchpath** or **!wmitrace.tmffile** to specify the TMF file before using **!wmitrace.dynamicprint** or **!wmitrace.logdump** to display trace buffer contents.

If you do not use either **!wmitrace.searchpath** or **!wmitrace.tmffile**, the system uses the value of the

TRACE_FORMAT_SEARCH_PATH environment variable. If that variable is not present, it uses the default.tmf file, which is included in Windows. If the system cannot find any formatting information for a trace message, it writes a "No format information found" error message in place of the trace message content.

!wmitrace.setprefix

9/20/2019 • 4 minutes to read • Edit Online

The **!wmitrace.setprefix** extension specifies the trace message prefix that is prepended to the trace messages from this session. This extension allows you to change the prefix during the debugging session.

```
!wmitrace.setprefix [+] PrefixVariables
!wmitrace.setprefix
```

Parameters

+
Causes *PrefixVariables* to be appended to the trace message prefix. If the **+** token is not used, *PrefixVariables* replaces the existing trace message prefix.

PrefixVariables
A set of variables that specifies the format and data in the trace message prefix.

The variables have the format %n!x!, where %n represents a data field and !x! represents the data type. You can also include separation characters, such as colons (:), semicolons (;), parentheses (( )), braces ({ }), and brackets ([ ]) to separate the fields.

Each %n variable represents a parameter that is described in the following table.

PREFIX VARIABLE IDENTIFIER	VARIABLE TYPE	DESCRIPTION
%1	string	The friendly name of the message GUID of the trace message. By default, the friendly name of a message GUID is the name of the directory in which the trace provider was built.
%2	string	Source file and line number. This variable represents the friendly name of the trace message. By default, the friendly name of a trace message is the name of the source file and the line number of the code that generated the trace message.
%3	ULONG	Thread ID. Identifies the thread that generated the trace message.
%4	string	Time stamp of the time that the trace message was generated.

PREFIX VARIABLE IDENTIFIER	VARIABLE TYPE	DESCRIPTION
%5	string	Kernel time. Displays the elapsed execution time for kernel-mode instruction, in CPU ticks, at the time that the trace message was generated.
%6	string	User time. Displays the elapsed execution time for user-mode instruction, in CPU ticks, at the time that the trace message was generated.
%7	LONG	Sequence number. Displays the local or global sequence number of the trace message. Local sequence numbers, which are unique only to this trace session, are the default.
%8	ULONG	Process ID. Identifies the process that generated the trace message.
%9	ULONG	CPU number. Identifies the CPU on which the trace message was generated.
%!FUNC!	string	Function name. Displays the name of the function that generated the trace message.
%!FLAGS%	string	Displays the name of the trace flags that enable the trace message.
%!LEVEL!	string	Displays the value of the trace level that enables the trace message.
%!COMPNAME!	string	Component name. Displays the name of the component of the provider that generated the trace message. The component name appears only if it is specified in the tracing code.

PREFIX VARIABLE IDENTIFIER	VARIABLE TYPE	DESCRIPTION
%!SUBCOMP!	string	Subcomponent name. Displays the name of the subcomponent of the provider that generated the trace message. The subcomponent name appears only if it is specified in the tracing code.

The symbol within exclamation marks is a conversion character that specifies the format and precision of the variable. For example, %8!04X! specifies the process ID formatted as a four-digit, unsigned, hexadecimal number.

DLL

This extension is exported by Wmitrace.dll.

This extension is available in Windows 2000 and later versions of Windows. If you want to use this extension with Windows 2000, you must first copy the Wmitrace.dll file from the winxp subdirectory of the Debugging Tools for Windows installation directory to the w2kfre subdirectory.

Example

The following command changes the trace message prefix in the debugger output to the following format:

!wmitrace.setprefix %2!s!: %!FUNC!: %8!04x!.%3!04x!: %4!s!:

This extension command sets the trace message prefix to the following format:

SourceFile_LineNumber: FunctionName: ProcessID.ThreadID: SystemTime:

As a result, the trace messages are prepended with the specified information in the specified format. The following code example is taken from a trace of the Tracedrv sample driver in the WDK.

```
tracedrv_c258: TracedrvDispatchDeviceControl: 0af4.0c64: 07/25/2003-13:55:39.998:  IOCTL = 1
```

Additional Information

For a conceptual overview of event tracing, see the Microsoft Windows SDK documentation. For information about trace message format files, see the "Trace Message Prefix" topic in the WDK documentation.

Remarks

When used with no parameters, **!wmitrace.setprefix** displays the current value of the trace message prefix.

The *trace message prefix* consists of data about the trace message that is prepended to each trace message during Windows software trace preprocessor (WPP) software tracing. This data originates in the trace log (.etl) file and the trace message format (.tmf) file. You can customize the format and data in trace message prefix.

The default trace message prefix is as follows:

```
[%9!d!]%8!04X!.%3!04X!::%4!s! [%1!s!]
```

and produces the following prefix:

```
[CPUNumber]ProcessID.ThreadID::SystemTime [ProviderDirectory]
```

You can change the format and data in the trace message prefix outside of the debugger by setting the

%TRACE_FORMAT_PREFIX% environment variable. For an example that illustrates how to set the trace message prefix outside of the debugger, see "Example 7: Customizing the Trace Message Prefix" in the Windows Driver Kit (WDK) documentation. If the trace message prefix of your messages varies from the default, this environment variable might be set on your computer.

The prefix that you set by using this extension command affects only the debugger output. The trace message prefix that appears in the trace log is determined by the default value and the value of the %TRACE_FORMAT_PREFIX% environment variable.

This extension is only useful during WPP software tracing, and earlier (legacy) methods of Event Tracing for Windows. Trace events that are produced by other manifested providers do not use trace message format (TMF) files, and therefore this extension does not affect them.

!wmitrace.start

12/5/2018 • 2 minutes to read • Edit Online

The **!wmitrace.start** extension starts the Event Tracing for Windows (ETW) logger on the target computer.

```
!wmitrace.start LoggerName [-cir Size | -seq Size] [-f File] [-b Size] [-max Num] [-min Num] [-kd] [-ft Time]
```

Parameters

LoggerName
Supplies a name to be used for the trace session. *LoggerName* cannot contain spaces or quotation marks.

-cir *Size*
Causes the log file to be written in a circular manner. *Size* specifies the maximum file size, in bytes. When the file reaches this length, new data will be written to the file in a circular manner, overwriting the file from beginning to end. This cannot be combined with the **-seq** parameter. If neither **-cir** nor **-seq** is specified, the file is written in buffered mode.

-seq *Num*
Causes the log file to be written in a sequential manner. *Size* specifies the maximum file size, in bytes. When the file reaches this length, the oldest data will be deleted from the beginning of the file whenever new data is appended to the end. This cannot be combined with the **-cir** parameter. If neither **-cir** nor **-seq** is specified, the file is written in buffered mode.

-f *File*
Specifies the name of the log file to be created on the target computer. *File* must include an absolute directory path, and cannot contain spaces or quotation marks.

-b *Size*
Specifies the size of each buffer, in kilobytes. The permissible range of *Size* is between 1 and 2048, inclusive.

-max *Num*
Specifies the maximum number of buffers to use. *Num* can be any positive integer.

-min *Num*
Specifies the minimum number of buffers to use. *Num* can be any positive integer.

-kd
Enables KD filter mode. Messages will be sent to the kernel debugger and displayed on the screen.

-ft *Time*
Specifies the duration of the flush timer, in seconds. Starting in Windows 8, you can specify the flush timer duration in milliseconds by appending **ms** to the *Time* value. For example, **-ft 100ms**.

Note If you start a tracing session in KD filter mode (**-kd**), trace buffers on the target computer are sent to the debugger on the host computer for display. This parameter specifies how often the buffers on the target computer are flushed and sent to the host computer.

DLL

This extension is exported by Wmitrace.dll.

This extension is available in Windows 7 and later versions of Windows.

Additional Information

For more details on the parameters of this extension, see StartTrace Function and EVENT_TRACE_PROPERTIES. For a conceptual overview of event tracing, see the Microsoft Windows SDK. For information about tracing tools, see the Windows Driver Kit (WDK).

Remarks

After using this extension, you must resume program execution (for example, by using the **g (Go)** command) in order for it to take effect. After a brief time, the target computer automatically breaks into the debugger again.

When the trace session is started, the system assigns it an ordinal number (the *logger ID*). The session can then be referred to either by the logger name or the logger ID.

To stop the ETW logger, use **!wmitrace.stop**.

!wmitrace.stop

12/5/2018 • 2 minutes to read • Edit Online

The **!wmitrace.stop** extension stops the Event Tracing for Windows (ETW) logger on the target computer.

```
!wmitrace.stop { LoggerID | LoggerName }
```

Parameters

LoggerID
Specifies the trace session. *LoggerID* is an ordinal number that the system assigns to each trace session on the computer.

LoggerName
Specifies the trace session. *LoggerName* is the text name that was specified when the trace session was started.

DLL

This extension is exported by Wmitrace.dll.

This extension is available in Windows 7 and later versions of Windows.

Additional Information

For a conceptual overview of event tracing, see the Microsoft Windows SDK. For information about tracing tools, see the Windows Driver Kit (WDK).

Remarks

After using this extension, you must resume program execution (for example, by using the **g (Go)** command) in order for it to take effect. After a brief time, the target computer automatically breaks into the debugger again.

To start the ETW logger, use **!wmitrace.start**.

!wmitrace.strdump

12/5/2018 • 2 minutes to read • Edit Online

The **!wmitrace.strdump** extension displays the WMI event trace structures. You can limit the display to the structures for a particular trace session.

```
!wmitrace.strdump [ LoggerID | LoggerName ]
```

Parameters

LoggerID
Limits the display to the event trace structures for the specified trace session. *LoggerID* specifies the trace session. It is an ordinal number that the system assigns to each trace session on the computer. If no parameter is specified, all trace sessions are displayed.

LoggerName
Limits the display to the event trace structures for the specified trace session. *LoggerName* is the text name that was specified when the trace session was started. If no parameter is specified, all trace sessions are displayed.

DLL

This extension is exported by Wmitrace.dll.

This extension is available in Windows 2000 and later versions of Windows. If you want to use this extension with Windows 2000, you must first copy the Wmitrace.dll file from the winxp subdirectory of the Debugging Tools for Windows installation directory to the w2kfre subdirectory.

Additional Information

For a conceptual overview of event tracing, see the Microsoft Windows SDK. For information about Tracelog, see the "Tracelog" topic in the Windows Driver Kit (WDK).

Remarks

To find the logger ID of a trace session, use the **!wmitrace.strdump** extension. Alternatively, you can use the Tracelog command tracelog -l to list the trace sessions and their basic properties, including the logger ID.

!wmitrace.tfp

12/5/2018 • 2 minutes to read • Edit Online

The **!wmitrace.tfp** extension command is obsolete. Use **!wmitrace.setprefix** instead.

!wmitrace.tmffile

12/5/2018 • 2 minutes to read • Edit Online

The **!wmitrace.tmffile** extension specifies a trace message format (TMF) file. The file specified by this extension is used to format trace messages displayed or written by other WMI tracing extensions.

```
!wmitrace.tmffile TMFFile
```

Parameters

TMFFile
Specifies a trace message format file.

DLL

This extension is exported by Wmitrace.dll.

This extension is available in Windows 2000 and later versions of Windows. If you want to use this extension with Windows 2000, you must first copy the Wmitrace.dll file from the winxp subdirectory of the Debugging Tools for Windows installation directory to the w2kfre subdirectory.

Additional Information

For a conceptual overview of event tracing, see the Microsoft Windows SDK. For information about trace message format files, see the "Trace Message Format File" topic in the Windows Driver Kit (WDK).

Remarks

Trace message format files (.tmf) are structured text files that are created during Windows software trace preprocessor (WPP) software tracing. These files contain instructions for formatting trace binary trace messages so that they can be displayed in human-readable form.

In order to display the trace message in a trace buffer (**!wmitrace.logdump**) or write them to a file (**!wmitrace.logsave**), you must first identify the TMF files for the trace messages.

You can use **!wmitrace.searchpath** to specify a directory in which TMF files are stored. The system then searches the directory for a TMF file that contains instructions for the messages that it is formatting. (It uses the message GUID to associate the message with the correct TMF file.)

However, you can use **!wmitrace.tmffile** to specify a particular TMF file. You must use **!wmitrace.tmffile** if the TMF file name is not a message GUID followed by the .tmf extension. Otherwise, the system will not find it.

If you do not use either **!wmitrace.searchpath** or **!wmitrace.tmffile**, the system uses the value of the TRACE_FORMAT_SEARCH_PATH environment variable. If that variable is not present, it uses the default.tmf file. If the system cannot find formatting information for a trace message, it writes a "No format information found" error message, instead of the trace message content.

Note If your driver uses UMDF version 1.11 or later, you do not need to use **!wmitrace.searchpath** or **!wmitrace.tmffile**.

This extension is only useful during WPP software tracing, and earlier (legacy) methods of Event Tracing for Windows. Trace events that are produced by other manifested providers do not use trace message format (TMF) files, and therefore this extension cannot be used with them.

!wmitrace.traceoperation

1/3/2019 • 2 minutes to read • Edit Online

The **!wmitrace.traceoperation** extension displays the progress messages from the tracing components in Windows.

```
!wmitrace.traceoperation {0 | 1 | 2}
```

Parameters

0
Disables the display of tracing progress messages.

1
Enables the display of tracing progress messages. All messages generated by the WMI tracing debugger extensions are displayed.

2
Enables the display of tracing progress messages. All messages generated by the WMI tracing debugger extensions or by Tracefmt are displayed.

DLL

This extension is exported by Wmitrace.dll.

Additional Information

For a conceptual overview of event tracing, see the Microsoft Windows SDK.

Remarks

This extension causes the tracing components to display verbose output. This feature is useful to troubleshoot software tracing.

OEM Support Extensions (kdex2x86.dll)

12/5/2018 • 2 minutes to read • Edit Online

The OEM Support Extensions can be found in the w2kfre\kdex2x86.dll subdirectory of the Debugging Tools for Windows installation directory.

You can use these extensions only with Windows 2000 targets.

These extensions are not described in this documentation. For descriptions of these extensions, see the OEM Support Tools documentation. To get this documentation, along with the entire OEM Support Tools package, go to Microsoft Support Article 253066 and follow the instructions on that page.

SieExtPub.dll

12/5/2018 • 2 minutes to read • Edit Online

The SieExtPub.dll extension module is not part of Microsoft Debugging Tools for Windows, and SieExtPub.dll is no longer included in Debug Diagnostic Tool. Many of the extension commands that were in SieExtPub.dll are now in ext.dll. To see a list of extension commands in ext.dll, enter the following command in the debugger.

.extmatch /e ext *

Note The **!critlist** extension command is no longer available.

Debugger-Related APIs

12/5/2018 • 2 minutes to read • Edit Online

This reference section includes:

Symbol Server API

The dbgeng.h Header File

The wdbgexts.h Header File

Symbol Server API

12/5/2018 • 2 minutes to read • Edit Online

The **SymbolServer*Xxx*** routines are needed if you want to write your own symbol server.

These routines are part of the DbgHelp interface. For full documentation regarding the DbgHelp and ImageHlp interfaces, see the "Debug Help Library" documentation (dbghelp.chm). This can be found in the sdk\help subdirectory of the Debugging Tools for Windows installation directory.

The dbgeng.h Header File

12/5/2018 • 2 minutes to read • Edit Online

Routines in the dbgeng.h header file are used to write DbgEng extensions.

For details on the dbgeng.h header file and information about how to write debugger extensions, see

For information about how to write debugger extensions, see Writing DbgEng Extensions and Using the Debugger Engine API.

The wdbgexts.h Header File

12/5/2018 • 2 minutes to read • Edit Online

Routines in the wdbgexts.h header file are used to write DbgEng extensions and WdbgExts extensions.

For details on the wdbgexts.h header file and information about how to write debugger extensions, see Writing WdbgExts Extensions.

Debugger Error and Warning Messages

12/5/2018 • 2 minutes to read • Edit Online

This reference section describes some of the error and warning messages that the debugger can display.

dbgerr001: PEB is Paged Out

12/5/2018 • 2 minutes to read • Edit Online

Debugger error **dbgerr001** displays the message "PEB is paged out." This error indicates that the process environment block (PEB) is not accessible.

To load symbols, the debugger looks at the list of modules loaded by the operating system. The pointer to the user-mode module list is one of the items stored in the PEB.

In order to reclaim memory, the Memory Manager may page out user-mode data to make space for other process or kernel mode components.

When this error occurs, it means that the debugger has detected that the PEB data structure for this process is no longer readable. Most likely, is has been paged out to disk.

Without this data structure, the debugger cannot determine what images the symbols should be loaded for.

Note This only affects symbol files for the user-mode modules. Kernel-mode modules and symbols are not affected, as they are tracked in a different list.

dbgerr002: Bad Pointer

12/5/2018 • 2 minutes to read • Edit Online

Debugger error **dbgerr002** displays the message "Bad pointer." This error indicates a problem retrieving a symbol file.

The symbol server has the file indexed, but is being redirected to another location to find the file. No file is accessible at this other location.

Two common causes of this problem are:

- The path is a UNC path, and the computer containing this server is not available.
- The path indicates a directory or file that has been deleted.

If your symbol store was created by using SymStore, you can find the full path to this file by examining file.ptr. For details, see Using SymStore.

dbgerr003: Mismatched Symbols

12/5/2018 • 2 minutes to read • Edit Online

Debugger error **dbgerr003** displays the message "*File* has mismatched symbols." This error indicates that DbgHelp found the symbol file represented by File, but that the symbol file is not the correct version, or that DbgHelp cannot confirm that the symbol file is the correct version.

The debugger might load the specified symbol file despite this error, depending on other requirements in the path.

dbgerr004: Page Not Present in Dump File

12/5/2018 • 2 minutes to read • Edit Online

Debugger error **dbgerr004** displays the message "Page *number* not present in dump file." This error indicates that the debugger needed a memory page that was not included in the dump file being debugged.

The specified *number* is the page frame number (PFN) corresponding to the location in the physical memory of the original page.

To suppress this error message, use the **.ignore_missing_pages 1** command. This command allows debugging to proceed, but does not display this error message.

Kernel-mode memory dumps come in three sizes, and the smaller sizes do not include all the memory pages. For details, see Varieties of Kernel-Mode Dump Files.

User-mode memory dumps also come in various sizes. See User-Mode Dump Files for details.

dbgerr005: Private Symbols Required

12/5/2018 • 2 minutes to read • Edit Online

Debugger error **dbgerr005** displays the message "Private symbols (symbols.pri) are required for locals." This error indicates that the debugger is unable to perform an action because private symbols are not present.

During kernel-mode debugging, the debugger needs symbols for Microsoft Windows. During user-mode debugging, the debugger needs symbols for the target application, and often needs symbols for Windows as well.

Some basic symbols, such as function names and global variables, are needed for even the most rudimentary debugging. These are called *public symbols*. Symbols such as data structure names, global variables visible in only one object file, local variables, and line number information are not always required for debugging, although they are useful for a more in-depth debugging session. These are called *private symbols*.

Many software manufacturers, including Microsoft, produce two versions of their symbol files. The version released to their customers contains only public symbols. The version used internally contains both public and private symbols.

Most debugging actions can be performed with public symbols alone. But certain actions -- such as displaying local variables -- require private symbols. When an action of this sort is attempted and private symbols are not available, this error message is displayed.

When you see this message, it is usually best to simply continue debugging. The information you were unable to obtain is probably not essential to properly debugging the target.

Stack Unwind Information Not Available

12/5/2018 • 2 minutes to read • Edit Online

When the debugger is examining the call stack, it may display the message "Stack unwind information not available. Following frames may be wrong."

This warning indicates that the debugger is not certain that the frames in the call stack listed after this message are correct.

To trace the call stack, the debugger examines the stack and analyzes the functions that have used the stack. This lets it identify the chain of return addresses that form the call stack. However, this procedure requires symbol information for each module containing the functions that used the stack.

If this symbol information is not available, the debugger is forced to make a best guess about which frames are return addresses. This warning information is displayed to indicate the uncertain nature of the call stack after this point.

No Header Information Available

12/5/2018 • 2 minutes to read • Edit Online

The debugger identifies the proper symbols by examining the headers of the relevant modules. If these module headers are paged out, the debugger (and the symbol server) are unable to find the proper symbols. When this occurs, "No Header Information Available" is displayed within the symbol error message.

For information about how to debug a target when module headers are paged out, see Reading Symbols from Paged-Out Headers.

WinDbg Graphical Interface Features

12/5/2018 • 2 minutes to read • Edit Online

This section discusses the elements of the WinDbg graphical user interface. These elements include the following:

File Menu

Edit Menu

View Menu

Debug Menu

Window Menu

Help Menu

Toolbar Buttons

Shortcut Keys

File Menu

12/5/2018 • 2 minutes to read • Edit Online

This section describes the following commands on the **File** menu of WinDbg:

File | Open Source File

File | Close Current Window

File | Open Executable

File | Attach to a Process

File | Open Crash Dump

File | Connect to Remote Session

File | Connect to Remote Stub

File | Kernel Debug

File | Symbol File Path

File | Source File Path

File | Image File Path

File | Open Workspace

File | Save Workspace

File | Save Workspace As

File | Clear Workspace

File | Delete Workspaces

File | Open Workspace in File

File | Save Workspace to File

File | Map Network Drive

File | Disconnect Network Drive

File | Recent Files

File | Exit

File | Open Source File

12/5/2018 • 2 minutes to read • Edit Online

Click **Open Source File** on the **File** menu to load a specific source file.

This command is equivalent to pressing CTRL+O or clicking the **Open source file (Ctrl+O)** button ( ).

Dialog Box

When you click **Open Source File**, the **Open Source File** dialog box appears. To open a file, do the following:

1. In the **Look in** list, select the directory where the file is located. The directory last opened is selected by default.

2. In the **Files of type** list, select the type of file that you want to open. Only files with the chosen extensions are displayed in the **Open Source File** dialog box. **Note** You can also use wildcard patterns in the **File name** box to display only files with a certain extension. The new wildcard pattern is retained in a session until you change it. You can use any combination of wildcard patterns, separated by semicolons. For example, entering ***.INC; *.H; *.CPP** displays all files with these extensions. The maximum number of characters in a line is 251.

3. If you find the file you want, double-click the file name, or click the file name and click **Open**.

   -OR-

   To discard changes and close the dialog box, click **Cancel**.

The names of the four files that you opened most recently in WinDbg are displayed when you point to **Recent files** on the **File** menu. To open one of these files, click its name.

Additional Information

For more information about source files and source paths and for other ways to load source files, see Source Path.

File | Close Current Window

12/5/2018 • 2 minutes to read • Edit Online

Click **Close Current Window** on the **File** menu to close the active debugging information window.

This command is equivalent to pressing CTRL+F4.

You can also close a debugging information window by clicking the **Close** button in the upper-right corner of the information window inside the WinDbg window.

File | Open Executable

12/5/2018 • 2 minutes to read • Edit Online

Click **Open Executable** on the **File** menu to start a new user-mode process and debug it.

This command is equivalent to pressing CTRL+E. You can use this command only when WinDbg is in dormant mode.

Dialog Box

When you click **Open Executable**, the **Open Executable** dialog box appears, and you can do the following:

- Enter the full path of the executable file in the **File name** box. Alternatively, you can use the dialog box to locate and select the proper file. You must specify the exact path to the executable file. Unlike the Microsoft Windows **Run** dialog box and a Command Prompt window, the **Open Executable** dialog box does not search the current path for an executable name.
- If you want to use command-line arguments with the executable file, enter them in the **Arguments** box.
- If you want to change the starting directory from the default directory enter the directory path in the **Start directory** box.
- If you want WinDbg to attach to any *child processes* (additional processes that the original target process started), select **Debug child processes also**.

After you make your selections, click **Open**.

Note When you use this command to open a source file, the path to that file is automatically appended to the source path.

If WinDbg is connected to a process server, you cannot use the **Open Executable** command.

Additional Information

For more information and other methods of starting new processes for debugging, see Debugging a User-Mode Process Using WinDbg.

File | Attach to a Process

12/5/2018 • 2 minutes to read • Edit Online

Click **Attach to a Process** on the **File** menu to debug a user-mode application that is currently running.

This command is equivalent to pressing F6. You can use this command only when WinDbg is in dormant mode.

Dialog Box

When you click **Attach to a Process**, the **Attach to Process** dialog box appears,and you can do the following:

- Select the line that contains the proper process ID and name (or enter the process ID in the **Process ID** box). **Note** Each listed process has an associated plus sign (**+**). You can click the plus sign to display information about that process' command line, services, and child processes.

  Note If WinDbg is connected to a process server, the **Attach to Process** dialog box will display processes that are running on the remote computer. For more information about process servers, see **Activating a Smart Client**.

- If you want to attach noninvasively to a process, select the **Noninvasive** check box.

After you make your selections, click **OK**.

Additional Information

For more information and other methods of attaching to a process, see Debugging a User-Mode Process Using WinDbg and Noninvasive Debugging (User Mode).

File | Open Crash Dump

12/5/2018 • 2 minutes to read • Edit Online

Click **Open Crash Dump** on the **File** menu to open a user-mode or kernel-mode crash dump file and to analyze it.

This command is equivalent to pressing CTRL+D. You can use this command only when WinDbg is in dormant mode.

Dialog Box

When you click **Open Crash Dump**, the **Open Crash Dump** dialog box appears. Enter the full path of the crash dump file in the **File name** box, or use the **Look in** list to find and select the proper path and file name. (Dump files typically end with the .dmp or .mdmp extension.)

After you choose the proper file, click **Open**.

Additional Information

For more information about analyzing crash dump files, see Analyzing a User-Mode Dump File or Analyzing a Kernel-Mode Dump File with WinDbg.

File | Connect to Remote Session

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

Click **Connect to Remote Session** on the **File** menu to make WinDbg a debugging client and to connect to an active debugging server.

This command is equivalent to pressing CTRL+R. You can use this command only when WinDbg is in dormant mode.

You cannot use this command to connect to a process server or a KD connection server; for that purpose, use File | Connect to Remote Stub instead.

Connect to Remote Debugger Session Dialog Box

When you click **Connect to Remote Session**, the **Connect to Remote Debugger Session** dialog box appears. You can use this dialog box to enter the remote connection parameters or to browse a list of debugging servers.

To manually specify the remote connection parameters, enter one of the following strings in the **Connection string** box:

```
npipe:server=Server,pipe=PipeName[,password=Password]

tcp:server=Server,port=Socket[,password=Password][,ipversion=6]

tcp:clicon=Server,port=Socket[,password=Password][,ipversion=6]

com:port=COMPort,baud=BaudRate,channel=COMChannel[,password=Password]

spipe:proto=Protocol,{certuser=Cert|machuser=Cert},server=Server,pipe=PipeName[,password=Password]

ssl:proto=Protocol,{certuser=Cert|machuser=Cert},server=Server,port=Socket[,password=Password]

ssl:proto=Protocol,{certuser=Cert|machuser=Cert},clicon=Server,port=Socket[,password=Password]
```

The various parameters in the preceding options have the following possible values:

Server
The network name of the computer on which the debugging server was created. Do not precede this name with backslashes (****).

PipeName
If you use the NPIPE or SPIPE protocol, *PipeName* is the name that was given to the pipe when the server was created.

Socket
If you use the TCP or SSL protocol, *Socket* is the same socket port number that was used when the server was created.

COMPort
If you use the COM protocol, *COMPort* specifies the COM port to use. The "COM" prefix is optional (for example, both "com2" and "2" are correct).

BaudRate
If you use the COM protocol, *BaudRate* should match the baud rate that you chose when the server was created.

COMChannel

If you use the COM protocol, *COMChannel* should match the channel number that you chose when the server was created.

Protocol
If you use the SSL or SPIPE protocol, *Protocol* should match the secure protocol that you used when the server was created.

Cert
If you use the SSL or SPIPE protocol, you should use the identical **certuser=***Cert* or **machuser=***Cert* parameter that was used when the server was created.

clicon
Specifies that the debugging server will try to connect to the client through a reverse connection. The client must use **clicon** if and only if the server is using **clicon**. In most cases, the debugging client is started before the debugging server when a reverse connection is used.

Password
If you used a password when the server was created, you must supply *Password* to create the debugging client. This value must match the original password. Passwords are case-sensitive. If the wrong password is supplied, the error message will specify "Error 0x80004005".

ipversion=6
(Debugging Tools for Windows 6.6.07 and earlier only) Forces the debugger to use IP version 6 rather than version 4 when you are using TCP to connect to the Internet. In Windows Vista and later versions, the debugger attempts to auto-default to IP version 6, making this option unnecessary.

Instead of manually specifying the remote connection parameters, you can press the **Browse** button in the **Connect to Remote Debugger Session** dialog box and use the **Browse Remote Servers** dialog box.

Browse Remote Servers Dialog Box

In the **Browse Remote Servers** dialog box, in the **Machine** text box, enter the name of the computer that the debugging server is running on. (The two initial backslashes are optional: "MyBox" and "\\MyBox" are both correct.) Then, press the **Refresh** button.

The **Servers** area lists all of the debugging servers that are running on that computer. Select any of the listed servers and then press ENTER or click **OK**. (You can also double-click one of the listed servers.) The proper connection string for the debugging server that you selected will now appear in the **Connection string** box in the **Connect to Remote Debugger Session** dialog box.

If the server is password-protected, the connection string includes **Password=**. *You must replace the asterisk (*****) with the actual password.

After you specify the server and password, click **OK** to open the connection.

This list of servers in the **Browse Remote Servers** dialog box can also include servers that no longer exist but were shut down improperly. If you connect to one of these nonexistent servers, you will receive an error message.

The list of servers does not include process servers and KD connection servers; you can list those servers only by using the File | Connect to Remote Stub command or by running **cdb -QR** *Server* from a Command Prompt window.

Additional Information

For more information and for other methods of joining a remote debugging session, see **Activating a Debugging Client**.

File | Connect to Remote Stub

12/5/2018 • 3 minutes to read • Edit Online

Click **Connect to Remote Stub** on the **File** menu to make WinDbg a smart client and connect to a process server or a KD connection server.

This command is equivalent to using the -premote command line option in user mode, or the -k kdsrv transport protocol command-line option in kernel mode. You can use this command only when WinDbg is in dormant mode.

You cannot use this command to connect to a debugging server; for that purpose, use File | Connect to Remote Session instead.

Connect to Remote Stub Server Dialog Box

When you click **Connect to Remote Stub**, the **Connect to Remote Stub Server** dialog box appears. You can use this dialog box to enter the remote connection parameters or to browse a list of process servers and KD connection servers.

To manually specify the remote connection parameters, enter one of the following strings in the **Connection string** box:

```
npipe:server=Server,pipe=PipeName[,password=Password]

tcp:server=Server,port=Socket[,password=Password][,ipversion=6]

tcp:clicon=Server,port=Socket[,password=Password][,ipversion=6]

com:port=COMPort,baud=BaudRate,channel=COMChannel[,password=Password]

spipe:proto=Protocol,{certuser=Cert|machuser=Cert},server=Server,pipe=PipeName[,password=Password]

ssl:proto=Protocol,{certuser=Cert|machuser=Cert},server=Server,port=Socket[,password=Password]

ssl:proto=Protocol,{certuser=Cert|machuser=Cert},clicon=Server,port=Socket[,password=Password]
```

The various parameters in the preceding options have the following possible values:

Server
The network name of the computer on which the process server or KD connection server was created. Do not precede this name with backslashes (****).

PipeName
If you use the NPIPE or SPIPE protocol, *PipeName* is the name that was given to the pipe when the server was created.

Socket
If you use the TCP or SSL protocol, *Socket* is the same socket port number that was used when the server was created.

COMPort
If you use the COM protocol, *COMPort* specifies the COM port to use. The "COM" prefix is optional (for example, both "com2" and "2" are correct).

BaudRate
If you use the COM protocol, *BaudRate* should match the baud rate that you chose when the server was created.

COMChannel
If you use the COM protocol, *COMChannel* should match the channel number that you chose when the server was created.

Protocol
(Windows 2000 and later) If you use the SSL or SPIPE protocol, *Protocol* should match the secure protocol that you used when the server was created.

Cert
(Windows 2000 and later) If you use the SSL or SPIPE protocol, you should use the identical **certuser=***Cert* or **machuser=***Cert* parameter that was used when the server was created.

clicon
Specifies that the process server or KD connection server will try to connect to the client through a reverse connection. The client must use **clicon** if and only if the server is using **clicon**. In most cases, the smart client is started before the server when a reverse connection is used.

Password
If you used a password when the server was created, you must supply *Password* to create the smart client. This value must match the original password. Passwords are case-sensitive. If the wrong password is supplied, the error message will specify "Error 0x80004005".

ipversion=6
(Debugging Tools for Windows 6.6.07 and earlier only) Forces the debugger to use IP version 6 rather than version 4 when you are using TCP to connect to the Internet. In Windows Vista and later versions, the debugger attempts to auto-default to IP version 6, making this option unnecessary.

Instead of manually specifying the remote connection parameters, you can press the **Browse** button in the **Connect to Remote Stub Server** dialog box and use the **Browse Remote Servers** dialog box.

Browse Remote Servers Dialog Box

In the **Browse Remote Servers** dialog box, in the **Machine** text box, enter the name of the computer that the process server or KD connection server is running on. (The two initial backslashes are optional: "MyBox" and "\\MyBox" are both correct.) Then, press the **Refresh** button.

The **Servers** area lists all of the process servers and KD connection servers that are running on that computer. Select any of the listed servers and then press ENTER or click **OK**. (You can also double-click one of the listed servers.) The proper connection string for the process server that you selected will now appear in the **Connection string** box in the **Connect to Remote Stub Server** dialog box.

If the server is password-protected, the connection string includes **Password=**. *You must replace the asterisk (*****) with the actual password.

After you specify the server and password, click **OK** to open the connection.

The list of servers in the **Browse Remote Servers** dialog box can also include servers that no longer exist but were shut down improperly. If you connect to one of these nonexistent servers, you will receive an error message.

The list of servers does not include debugging servers. To view those servers, use the File | Connect to Remote Session command.

Additional Information

For more information and for other methods of joining a remote stub session, see **Activating a Smart Client** and **Activating a Smart Client (Kernel Mode)**.

File | Kernel Debug

12/5/2018 • 2 minutes to read • Edit Online

Click **Kernel Debug** on the **File** menu to debug a target computer in kernel mode.

This command is equivalent to pressing CTRL+K. You can use this command only when WinDbg is in dormant mode.

Dialog Box

When you click **Kernel Debug**, the **Kernel Debugging** dialog box appears with these tabs:

- The **COM** tab indicates that the connection will use a COM port. In the **Baud Rate** box, enter the baud rate. In the **Port** box, enter the name of the COM port. For more information, see Setting Up a Serial Connection Manually.

  You can also use the COM tab to connect to virtual machine through a named pipe. In the **Port** box, enter *****VMHost***\pipe***PipeName*. *VMHost* specifies the name of the physical computer on which the virtual machine is running. If the virtual machine is running on the same computer as the kernel debugger itself, use a single period (.) for *VMHost*. For more information, see Setting Up a Connection to a Virtual Machine.

- The **1394** tab indicates that the connection will use 1394. In the **Channel** box, enter the 1394 channel number. 1394 debugging is supported only if both the host computer and target computer are running Windows XP or later versions of the Windows operating system. For more information, see Setting Up a 1394 Connection Manually.

- The **USB** tab indicates that the connection will use USB 2.0 or USB 3.0. In the **Target Name** box, enter the target name that you created when you configured the target computer. For more information, see Setting Up a USB 2.0 Connection Manually and Setting Up a USB 3.0 Connection Manually.

- The **NET** tab indicates that the connection will use Ethernet. In the **Port Number** box, enter the port number that you specified when you configured the target computer. In the **Key** box, enter the key that was generated for you (or that you created) when you configured the target computer. For more information, see Setting Up a Network Connection Manually.

- The **Local** tab indicates that WinDbg will perform local kernel debugging. Local kernel debugging is supported only on Windows XP and later.

For more information and for other methods of beginning a kernel debugging session, see Live Kernel-Mode Debugging Using WinDbg.

File | Symbol File Path

12/5/2018 • 2 minutes to read • Edit Online

Click **Symbol File Path** on the **File** menu to display, set, or append to the symbol path.

This command is equivalent to pressing CTRL+S.

Symbol Search Path Dialog Box

When you click **Symbol File Path**, the **Symbol Search Path** dialog box appears. This dialog box displays the current symbol path. If the **Symbol path** box is blank, there is no current symbol path.

You can enter a new path or edit the old path. If you want to search more than one directory, separate the directory names with semicolons.

Click **OK** to save changes, or click **Cancel** to discard changes.

If you select the **Reload** check box, the debugger will reload all loaded symbols and images after you click **OK**. The **Reload** command is equivalent to using the **.reload (Reload Module)** command.

You can also click **Browse** to open the **Browse For Folder** dialog box.

Browse For Folder Dialog Box

In the **Browse For Folder** dialog box, you can browse through the folders on your computer or your network. You can also click the **Make New Folder** button to create a new folder. If you right-click a file or folder in this dialog box, a standard Windows shortcut menu appears.

Click **OK** to append the selected folder to the symbol path and return to the **Symbol Search Path** dialog box.

Additional Information

For more information and for other ways to change the symbol path, see Symbol Path.

File | Source File Path

12/5/2018 • 2 minutes to read • Edit Online

Click **Source File Path** on the **File** menu to display, set, or append to the source path.

This command is equivalent to pressing CTRL+P.

Source Search Path Dialog Box

When you click **Source File Path**, the **Source Search Path** dialog box appears. This dialog box displays the current source path. If the **Source path** box is blank, there is no current source path.

You can enter a new path or edit the old path. If you want to search more than one directory, separate the directory names with semicolons.

If you are performing remote debugging, the **Local** check box will be available. Select this box to edit your debugging client's local source path; clear it to edit the debugging server's source path.

Click **OK** to save changes, or click **Cancel** to discard changes.

You can also click **Browse** to open the **Browse For Folder** dialog box.

Browse For Folder Dialog Box

In the **Browse For Folder** dialog box, you can browse through the folders on your computer or your network. You can also click the **Make New Folder** button to create a new folder. If you right-click a file or folder in this dialog box, a standard Windows shortcut menu appears.

Click **OK** to append the selected folder to the source path and return to the **Source Search Path** dialog box.

Additional Information

For more information and for other ways to change this path, see Source Path.

File | Image File Path

12/5/2018 • 2 minutes to read • Edit Online

Click **Image File Path** on the **File** menu to display, set, or append to the executable image path.

This command is equivalent to pressing CTRL+I.

Executable Image Search Path Dialog Box

When you click **Image File Path**, the **Executable Image Search Path** dialog box appears. This dialog box displays the current executable image path. If the **Image path** box is blank, there is no current executable image path.

You can enter a new path or edit the old path. If you want to search more than one directory, separate the directory names with semicolons.

Click **OK** to save changes, or click **Cancel** to discard changes.

If you select the **Reload** check box, the debugger will reload all loaded image and symbol files after you click **OK**. This command is equivalent to using the **.reload (Reload Module)** command.

You can also click **Browse** to open the **Browse For Folder** dialog box.

Browse For Folder Dialog Box

In the **Browse For Folder** dialog box, you can browse through the folders on your computer or your network. You can also use the **Make New Folder** button to create a new folder. If you right-click a file or folder in this dialog box, a standard Windows shortcut menu appears.

Click **OK** to append the selected folder to the executable image path and return to the **Executable Image Search Path** dialog box.

File | Open Workspace

12/5/2018 • 2 minutes to read • Edit Online

Click **Open Workspace** on the **File** menu to open a saved workspace.

This command is equivalent to pressing CTRL+W.

Dialog Box

When you click **Open Workspace**, the **Open Workspace** dialog box appears. This dialog box contains a list of all named workspaces in the **Workspaces** area. Default or implicit workspaces are not listed since opening them directly will cause problems with the implicit ordering. If you want to open an implicit workspace directly, you must save it explicitly. For more information on named and default workspaces, see Creating and Opening a Workspace.

Enter the name of the workspace that you want to open in the **Workspace** box or select the workspace name in the **Workspaces** area. Then, click **OK** to open the selected workspace, or click **Cancel** to return the debugger to its previous state.

Additional Information

For more information about the different levels of workspaces and how to use them, see Using Workspaces.

File | Save Workspace

12/5/2018 • 2 minutes to read • Edit Online

Click **Save Workspace** on the **File** menu to save the current workspace under its current workspace name.

Additional Information

For more information about the different levels of workspaces and how to use them, see Using Workspaces.

File | Save Workspace As

12/5/2018 • 2 minutes to read • Edit Online

Click **Save Workspace As** on the **File** menu to save the current workspace under a new workspace name.

Dialog Box

When you click **Save Workspace As**, the **Save Workspace As** dialog box appears. This dialog box contains a list of all existing workspace names in the **Workspaces** area. The name of the current workspace is shown in the **Workspace** box.

Enter the name that you want to use to save the workspace in the **Workspace** box, select the workspace name in the **Workspaces** area, or just leave the current name as it exists.

Click **OK** to save the workspace, or click **Cancel** to not save the workspace.

Additional Information

For more information about the different levels of workspaces and how to use them, see Using Workspaces.

File | Clear Workspace

12/5/2018 • 2 minutes to read • Edit Online

Click **Clear Workspace** on the **File** menu to erase items in the current workspace.

Dialog Box

When you click **Clear Workspace**, the **Clear Workspace** dialog box appears. This dialog box contains a list of all of the items that are contained in the current workspace in the **Items in Workspace** area.

Use the **Clear** and **Clear All** buttons to remove items from the **Items in Workspace** area. If you make an error, use the **Save** and **Save All** buttons to return items to this list.

Click **OK** to make these changes, or click **Cancel** to discard these changes.

Additional Information

For more information about the different levels of workspaces and how to use them, see Using Workspaces.

File | Delete Workspaces

12/5/2018 • 2 minutes to read • Edit Online

Click **Delete Workspaces** on the **File** menu to delete one or more existing workspaces.

Dialog Box

When you click **Delete Workspaces**, the **Delete Workspaces** dialog box appears. In this dialog box, select the workspace that you want to delete and click **Delete**.

Click **Close** to close the dialog box.

Additional Information

For more information about the different levels of workspaces and how to use them, see Using Workspaces.

File | Open Workspace in File

12/5/2018 • 2 minutes to read • Edit Online

Click **Open Workspace in File** on the **File** menu to load a workspace that was previously saved to a file.

Dialog Box

When you click **Open Workspace in File**, the **Open Workspace in File** dialog box appears. In this dialog box, enter the name of the file that you want to load, or use the **Look in** list to navigate to the file and select it. (Workspace files should use the .wew extension.)

Click **OK** to load the workspace, or click **Cancel** to close the dialog box.

Additional Information

For more information about the different levels of workspaces and how to use them, see Using Workspaces.

File | Save Workspace to File

12/5/2018 • 2 minutes to read • Edit Online

Click **Save Workspace to File** on the **File** menu to save the current workspace to a file.

Dialog Box

When you click **Save Workspace to File**, the **Save Workspace to File** dialog box appears. In this dialog box, enter the name of the file that you want to save the workspace as. Then, use the **Save in** list to navigate to the directory where you want to save the file, or select a specific file you want to overwrite. (The default file extension is .wew.)

Click **OK** to save the file, or click **Cancel** to exit.

Additional Information

For more information about the different levels of workspaces and how to use them, see Using Workspaces.

File | Map Network Drive

12/5/2018 • 2 minutes to read • Edit Online

Click **Map Network Drive** on the **File** menu to add network connections and assign drive letters to these connections.

Dialog Box

When you click **Map Network Drive**, the **Map Network Drive** dialog box appears. Use the **Drive** and **Folder** menus to choose a server and share and assign a drive letter to it.

This dialog box works exactly like the corresponding feature of Windows Explorer.

The **File | Map Network Drive** command affects only the network connections of the computer on which WinDbg is running. If you are using WinDbg as a client in a remote debugging session and you want to change the network connections of the server, you must use a **.shell net use** command.

Additional Information

For more information about accessing the command shell, see Using Shell Commands.

File | Disconnect Network Drive

12/5/2018 • 2 minutes to read • Edit Online

Click **Disconnect Network Drive** on the **File** menu to remove network connections.

Dialog Box

When you click **Disconnect Network Drive**, the **Disconnect Network Drives** dialog box appears. In the **Network Drives** box, select the connection you want to remove and click **OK**.

This dialog box works exactly like the corresponding feature of Windows Explorer.

The **File | Disconnect Network Drive** command affects only the network connections of the computer on which WinDbg is running. If you are using WinDbg as a client in a remote debugging session and you want to change the network connections of the server, you must use a **.shell net use** command.

Additional Information

For more information about accessing the command shell, see Using Shell Commands.

File | Recent Files

12/5/2018 • 2 minutes to read • Edit Online

Point to **Recent Files** on the **File** menu to display a list of the four source files that you most recently opened in WinDbg.

To open one of these files, click its name from the menu.

Additional Information

For more information about source files and source paths, and for other ways to load source files, see Source Path.

File | Exit

12/5/2018 • 2 minutes to read • Edit Online

Click **Exit** on the **File** menu to end the debugging session and exit WinDbg.

This command is equivalent to pressing ALT+F4.

Additional Information

For more information about exiting WinDbg or ending the debugging session, see Ending a Debugging Session in WinDbg.

Edit Menu

12/5/2018 • 2 minutes to read • Edit Online

This section describes the following commands on the **Edit** menu of WinDbg:

Edit | Cut

Edit | Copy

Edit | Copy Formatted

Edit | Paste

Edit | Select All

Edit | Write Window Text to File

Edit | Copy Window Text to Clipboard

Edit | Add to Command Output

Edit | Clear Command Output

Edit | Evaluate Selection

Edit | Display Selected Type

Edit | Find

Edit | Find Next

Edit | Go to Address

Edit | Go to Line

Edit | Go to Current Instruction

Edit | Set Current Instruction

Edit | Breakpoints

Edit | Open/Close Log File

Edit | Cut

12/5/2018 • 2 minutes to read • Edit Online

Click **Cut** on the **Edit** menu to delete any text that you have selected and to move it to the clipboard.

This command is equivalent to pressing CTRL+X or SHIFT+DELETE or clicking the **Cut (Ctrl+X)** button ( ) on the toolbar.

You can use the **Cut** command on the **Edit** menu only with docked or tabbed windows, but you can use the shortcut keys and the toolbar button with any window that supports this feature.

Additional Information

For more information about how to select, copy, cut, and paste text and about how these operations vary from window to window, see Cutting and Pasting Text.

Edit | Copy

12/5/2018 • 2 minutes to read • Edit Online

Click **Copy** on the **Edit** menu to copy to the clipboard any text that you have selected.

This command is equivalent to pressing CTRL+C or CTRL+INSERT or clicking the **Copy (Ctrl+C)** button ( ) on the toolbar.

You can use the **Copy** command only with docked or tabbed windows. You can use the shortcut keys and the toolbar button with any window that supports this feature.

Additional Information

For more information about how to select, copy, cut, and paste text and about how these operations vary from window to window, see Cutting and Pasting Text.

Edit | Paste

12/5/2018 • 2 minutes to read • Edit Online

Click **Paste** on the **Edit** menu to paste text from the clipboard to the current cursor location.

This command is equivalent to pressing CTRL+V or SHIFT+INSERT or clicking the **Paste (Ctrl+V)** button ( ) on the toolbar.

You can use the **Paste** command only with docked or tabbed windows. But you can use the shortcut keys and the toolbar button with any window that supports this feature.

Additional Information

For more information about how to select, copy, cut, and paste text and about how these operations vary from window to window, see Cutting and Pasting Text.

Edit | Select All

12/5/2018 • 2 minutes to read • Edit Online

Click **Select All** on the **Edit** menu to select all of the text in the active Debugger Command window, Disassembly window, Source window, or dialog box.

This command is equivalent to pressing CTRL+A.

Additional Information

For more information about how to select, copy, cut, and paste text and about how these operations vary from window to window, see Cutting and Pasting Text.

Edit | Write Window Text to File

12/5/2018 • 2 minutes to read • Edit Online

Click **Write Window Text to File** on the **Edit** menu to save all of the text in the active debugging information window to a file.

This command is available only if the active window is the Debugger Command window, Calls window, or Scratch Pad.

Dialog Box

When you click **Write Windows Text to File**, the **Write Window Text to File** dialog box appears. In this dialog box, enter the name of the file where you want to save the window text. You can browse in the **Save in** list to the directory that you want or select a specific file that you want to overwrite. The default file name extension is .txt.

Click **Save** to save the file or click **Cancel** to exit.

Additional Information

For more information about how to select, copy, cut, and paste text and about how these operations vary from window to window, see Cutting and Pasting Text.

Edit | Add to Command Output

12/5/2018 • 2 minutes to read • Edit Online

Click **Add to Command Output** on the **Edit** menu to insert a comment into the Debugger Command window.

Type a comment into the **Text** box and then click **OK**.

Your comment appears in the Debugger Command window and in any open log file. However, the comment does not appear in any Windows debuggers that are remotely connected to your session.

Additional Information

For more information about other features of the Debugger Command window, see Using Debugger Commands.

Edit | Clear Command Output

12/5/2018 • 2 minutes to read • Edit Online

Click **Clear Command Output** on the **Edit** menu to clear all of the text from the Debugger Command window and clear the command history.

Additional Information

For more information about other features of the Debugger Command window, see Using Debugger Commands.

Edit | Evaluate Selection

12/5/2018 • 2 minutes to read • Edit Online

Click **Evaluate Selection** on the **Edit** menu to evaluate the current selection in the Source window and display the result in the Debugger Command window.

This command is equivalent to pressing CTRL+SHIFT+V, clicking **Evaluate selection** on the Source window's shortcut menu, or using the **?? (Evaluate C++ Expression)** command together with the selected text as its argument.

If the selected text includes more than one line, a syntax error results. If no text is selected in a Source window, you cannot use this command.

Additional Information

For more information about other features of Source windows, see Source Window.

Edit | Display Selected Type

12/5/2018 • 2 minutes to read • Edit Online

Click **Display Selected Type** on the **Edit** menu to determine the data type of the current selection in the Source window and to display the type in the Debugger Command window.

This command is equivalent to pressing CTRL+SHIFT+Y or clicking **Display selected type** on the Source window's shortcut menu.

If the selected text includes more than a single object, a syntax error or other irregular results might be displayed. If no text is selected in a Source window, you cannot use this command.

Additional Information

For more information about other features of Source windows, see Source Window.

Edit | Find

12/5/2018 • 2 minutes to read • Edit Online

Click **Find** on the **Edit** menu to find text in the active debugging information window.

Note The active window must be the Debugger Command window or a Source window.

This command is equivalent to pressing CTRL+F.

Dialog Box

When you click **Find**, the **Find** dialog box appears. In this dialog box, in the **Find what** box, enter the text that you want to find. If there is already text selected, this text automatically appears in the **Find what** box.

In the **Direction** area, click **Up** or **Down** to specify the direction of your search. The search begins wherever the cursor is in the window. You can put the cursor at any location by using the mouse pointer.

Select **Match whole word only** if you want to search for a single whole word. (If you select this option when you search for multiple words, you always receive a failed search.))

Select **Match case** to perform a case-sensitive search.

The **Find** command only changes the WinDbg display. This command does not affect the execution of the target or any other debugger operations.

After you close the **Find** dialog box, you can repeat the search in a forward direction by using the Edit | Find Next command or pressing F3. You can repeat the search in a backward direction by pressing SHIFT+F3.

Additional Information

For more information about other ways to find text in debugging information windows, see Moving Through a Window.

Edit | Find Next

12/5/2018 • 2 minutes to read • Edit Online

Click **Find Next** on the **Edit** menu to repeat the previous search and find the next match.

This command is equivalent to pressing F3.

To repeat the search in a backward direction, press SHIFT+F3.

If you have not previously performed any searches, use the **Edit | Find Next** command, press F3, or press SHIFT+F3 to open the **Find** dialog box (similar to the Edit | Find command).

Additional Information

For more information about other ways of finding text in debugging information windows, see Moving Through a Window.

Edit | Go to Address

12/5/2018 • 2 minutes to read • Edit Online

Click **Go to Address** on the **Edit** menu to go to an address in the target's virtual address space.

This command is equivalent to pressing CTRL+G.

Dialog Box

When you click **Go to Address**, the **View Code Offset** dialog box appears. In this dialog box, enter the address that you want to move to. This address can be an expression (such as a function, symbol, or integer memory address) or any valid address expression. If the address is ambiguous, the dialog box displays a list that contains all of the ambiguous items.

Note If you put the cursor on a line within the Disassembly window or a Source window and then use the**Go to Address** command, the address of the line that you have selected will appear in the **View Code Offset** dialog box. You can use this address or replace it with any address of your choice.

After you click **OK**, the debugger moves the caret (^) to the beginning of the function or address in the Disassembly window or a Source window.

You can use the **Go to Address** command in any window that is currently active. If the debugger is in disassembly mode, WinDbg finds the address in the Disassembly window. If the debugger is in source mode, WinDbg finds the address in a Source window. If the address cannot be found in a Source window, WinDbg finds it in the Disassembly window. If the appropriate window is not open, WinDbg opens it.

The **Go to Address** command only changes the WinDbg display. This command does not affect the execution of the target or any other debugger operations.

Additional Information

For more information about other ways of finding text in debugging information windows, see Moving Through a Window.

Edit | Go to Line

12/5/2018 • 2 minutes to read • Edit Online

Click **Go to Line** on the **Edit** menu to search for a specific line in the currently-active Source window. If the active window is not a Source window, this command has no effect.

This command is equivalent to pressing CTRL+L.

Dialog Box

When you click **Go to Line**, the **Go to Line** dialog box appears. In this dialog box, enter the line number that you want to find and then click **OK**. The debugger will move the caret (^) to that line. If the line number is bigger than the last line in the file, the cursor will move to the end of the file.

The **Go to Line** command only changes the WinDbg display. This command does not affect the execution of the target or any other debugger operations.

Additional Information

For more information about other ways of finding text in debugging information windows, see Moving Through a Window.

Edit | Go to Current Instruction

12/5/2018 • 2 minutes to read • Edit Online

Click **Go to Current Instruction** on the **Edit** menu to open the debugging information window that contains the current instruction and to highlight this instruction.

This command is equivalent to pressing ALT+ASTERISK (using the ASTERISK key on the numeric keypad).

If the current instruction corresponds to a known source file, WinDbg opens the Source window that contains this source file. If no such window exists, WinDbg opens one. The current line is highlighted.

If the current instruction is not in a known source file and the Disassembly window is open, WinDbg opens the Disassembly window and the current line is highlighted. However, if the Disassembly window is closed, the **Go to Current Instruction** command does not open it.

This command only changes the WinDbg display. This command does not affect the execution of the target or any other debugger operations.

Edit | Set Current Instruction

12/5/2018 • 2 minutes to read • Edit Online

Click **Set Current Instruction** on the **Edit** menu to change the value of the instruction pointer to the instruction that corresponds to the current line in the active Source window.

This command is equivalent to pressing CTRL+SHIFT+I or clicking **Set instruction pointer to current line** on the Source window's shortcut menu.

Edit | Breakpoints

12/5/2018 • 4 minutes to read • Edit Online

Click **Breakpoints** on the **Edit** menu to display or control breakpoints.

This command is equivalent to pressing ALT+F9. If a source window or the disassembly window is not active, you can also press f9 or click the **insert or remove breakpoint (f9)** button ( ) on the toolbar.

However, if a Source window or the Disassembly window is open, the **Insert or remove breakpoint (F9)** button and the F9 key set a breakpoint on the current line. (If there is already a breakpoint set at the current line, this button or key will remove the breakpoint.)

If a statement or call spans multiple lines, WinDbg sets the breakpoint on the last line of the statement or call. You should insert the caret (^) on or before the statement to set a breakpoint for the whole statement. If the debugger cannot set a breakpoint at the current caret position, it will search in a downward direction for the next allowed position and insert the breakpoint there.

Dialog Box

When you click **Breakpoints**, the **Breakpoints** dialog box appears. This dialog box displays all current breakpoint information and is presented in the following columns:

- The breakpoint number. This number is a decimal number that you can use to refer to the breakpoint in future commands.
- The breakpoint status. This status can be **e** (enabled) or **d** (disabled).
- (Unresolved breakpoints only) The letter **u**. This letter appears if the breakpoint is unresolved (that is, it does not match any currently-loaded module address). For details, see Unresolved Breakpoints (bu Breakpoints).
- The virtual address of the breakpoint. If you have enabled the loading of source line numbers, the display includes file and line number information instead of address offsets. If the breakpoint is unresolved, the address appears at the end of the listing instead of here.
- (Processor breakpoints only) Type and size information. This information can be **e** (execute), **r** (read/write), **w** (write), or **i** (input/output). These types are followed with the size of the block, in bytes. For details, see Processor Breakpoints (ba Breakpoints).
- The number of passes that are remaining until the breakpoint becomes active, followed by the initial number of passes in parentheses. The number of times that the program counter passes through the breakpoint without breaking is one less than the value of this number. Therefore, this number is never lower than 1. Note also that this number counts only the times the application executes through this point. In other words, stepping over this point does not count. After the full count has been reached, you can reset the count only by clearing and resetting the breakpoint.
- The associated process and thread. If thread is given with three asterisks (***), this breakpoint is not a thread-specific breakpoint.
- The module and function, with offset, that correspond to the breakpoint address. If the breakpoint is unresolved, the breakpoint address appears here, in parentheses. If the breakpoint is set on a valid address but symbol information is missing, this column will be blank.
- The command string that is automatically executed when this breakpoint is hit. This command string is displayed in quotation marks. If the breakpoint is hit, the commands in this command string are executed

until application execution resumes. Any command that resumes program execution (such as **g** or **t**) will stop the execution of the command list.

If you select any breakpoint, you can then click the **Remove**, **Disable**, or **Enable** button. The **Remove** button permanently removes the breakpoint. The **Disable** button temporarily deactivates the breakpoint. The **Enable** button re-enables a disabled breakpoint.

The **Remove All** button permanently removes all breakpoints.

You can also enter commands in the **Command** box in the following ways:

- If you enter a **bp (Set Breakpoint)**, **bu (Set Unresolved Breakpoint)**, **bm (Set Symbol Breakpoint)**, **ba (Break on Access)**, **bc (Breakpoint Clear)**, **bd (Breakpoint Disable)**, or **be (Breakpoint Enable)** command, the **Command** box works as if you were entering the command in the Debugger Command window. However, the command itself must be in lowercase letters. The command cannot begin with a thread specifier. If you want to use a thread specifier, enter it in the **Thread** box without the initial tilde (~).
- If you enter any other text, the text will be treated as the argument string for a **bu (Set Unresolved Breakpoint)** command. That is, the debugger prefixes your entry with **bu** and a space and then executes it as a command.

When you are entering a new breakpoint, you can also do the following:

- Create a thread-specific breakpoint by entering a thread specifier in the **Thread** box. Do not include the tilde (~) character that is typically prefixed to a thread specifier.
- Create a conditional breakpoint by entering a condition in the **Condition** box. The condition can be any evaluable expression, and it will be evaluated according to the current expression syntax (see Evaluating Expressions). For more information about these types of breakpoints, see Setting a Conditional Breakpoint.

Additional Information

For more information about how to use breakpoints, other breakpoint commands and methods of controlling breakpoints, and setting breakpoints in user space from a kernel debugger, see Using Breakpoints. For more information about conditional breakpoints, see Setting a Conditional Breakpoint.

Edit | Open/Close Log File

12/5/2018 • 2 minutes to read • Edit Online

Click **Open/Close Log File** on the **Edit** menu to write to a new log file, append to an existing log file, or close an open log file.

Dialog Box

When you click **Open/Close Log File**, the **Open/Close Log File** dialog box appears. This dialog box displays the current log file, if one is already open.

If the **Log file name** box is blank, you can enter a log file name. If this file already exists, WinDbg overwrites the existing file, unless you select the **Append** check box. If you specify a file name but no path, WinDbg puts the file in the default directory that you started WinDbg from.

If the **Log file name** box already displays a file name, you can click **Close Open Log File** to close the file. If you clear the **Log file name** box and enter a new log file name, the previous log file will be closed.

Click **OK** to save changes, or click **Cancel** to discard changes.

If you click **OK** when no log file name appears in the **Log file name** box, it has no effect. That is, WinDbg does not close a log file or open a log file.

However, if a log file is already active and you click **OK** without clearing its name or selecting **Append**, WinDbg deletes the log file and uses a new file that has the same name.

View Menu

12/5/2018 • 2 minutes to read • Edit Online

This section describes the following commands on the **View** menu of WinDbg:

View | Command

View | Watch

View | Locals

View | Registers

View | Memory

View | Call Stack

View | Disassembly

View | Scratch Pad

View | Processes and Threads

View | Command Browser

View | Recent Commands

View | Set Browser Start Command

View | Verbose Output

View | Event Timestamps

View | Show Version

View | Toolbar

View | Status Bar

View | Font

View | Options

View | Source language file extensions

View | WinDbg Command Line

View | Command

12/5/2018 • 2 minutes to read • Edit Online

Click **Command** on the **View** menu to open the Debugger Command window. If this window is already open, it becomes active.

This command is equivalent to pressing ALT+1 or clicking the **Command (Alt+1)** button ( ) on the toolbar.

For more information about this window and its uses, see Debugger Command Window.

View | Watch

12/5/2018 • 2 minutes to read • Edit Online

Click **Watch** on the **View** menu to open the Watch window. If this window is already open, it becomes active.

This command is equivalent to pressing ALT+2 or clicking the **Watch (Alt+2)** button ( ) on the toolbar.

View | Locals

12/5/2018 • 2 minutes to read • Edit Online

Click **Locals** on the **View** menu to open the Locals window. If this window is already open, it becomes active.

This command is equivalent to pressing ALT+3 or clicking the **Locals (Alt+3)** button ( ) on the toolbar.

For more information about this window and its uses, see Locals Window.

View | Registers

12/5/2018 • 2 minutes to read • Edit Online

Click **Registers** on the **View** menu to open the Registers window. If this window is already open, it becomes active.

This command is equivalent to pressing ALT+4 or clicking the **Registers (Alt+4)** button ( ) on the toolbar.

For more information about this window and its uses, see Registers Window.

View | Memory

12/5/2018 • 2 minutes to read • Edit Online

Click **Memory** on the **View** menu to open a new Memory window.

Note You can have multiple Memory windows open at the same time. Each window can display a different region of memory. Only the Memory window and the Source window have this ability. All other debugging information windows are limited to a single instance.

The **View** command is equivalent to pressing ALT+5 or clicking the **Memory window (Alt+5)** button ( ) on the toolbar.

For more information about this window and its uses, see Memory Window.

View | Call Stack

12/5/2018 • 2 minutes to read • Edit Online

Click **Call Stack** on the **View** menu to open the Calls window. If this window is already open, it becomes active.

This command is equivalent to pressing ALT+6 or clicking the **Call stack (Alt+6)** button ( ) on the toolbar.

For more information about this window and its uses, see Calls Window.

View | Disassembly

12/5/2018 • 2 minutes to read • Edit Online

Click **Disassembly** on the **View** menu to open the Disassembly window. If this window is already open, it becomes active.

This command is equivalent to pressing ALT+7 or clicking the **Disassembly (Alt+7)** button ( ) on the toolbar.

For more information about this window and its uses, see Disassembly Window.

View | Scratch Pad

12/5/2018 • 2 minutes to read • Edit Online

Click **Scratch Pad** on the **View** menu to open the Scratch Pad. If this window is already open, it becomes active.

This command is equivalent to pressing ALT+8 or clicking the **Scratch Pad (Alt+8)** button ( ) on the toolbar.

View | Processes and Threads

12/5/2018 • 2 minutes to read • Edit Online

Click **Processes and Threads** on the **View** menu to open the Processes and Threads window. If this window is already open, it becomes active.

This command is equivalent to pressing ALT+9 or clicking the **Processes and Threads (Alt+9)** button ( ) on the toolbar.

For more information about this window and its uses, see Processes and Threads Window.

View | Command Browser

12/5/2018 • 2 minutes to read • Edit Online

Click **Command Browser** on the **View** menu to open the Command Browser window. If this window is already open, it becomes active.

This command is equivalent to pressing CTRL+N or clicking the **Command Browser (Ctrl+N)** button ( ) on the toolbar.

For more information about this window and its uses, see Command Browser Window.

View | Verbose Output

12/5/2018 • 2 minutes to read • Edit Online

Click **Verbose Output** on the **View** menu to turn verbose mode on and off.

This command is equivalent to pressing CTRL+ALT+V. (and to pressing CTRL+V in KD).

When verbose mode is turned on, some display commands (such as register dumping) produce more detailed output. Every MODULE LOAD operation that is sent to the debugger is displayed. And every time that a driver or DLL is loaded by the operating system, the debugger is notified.

View | Show Version

12/5/2018 • 2 minutes to read • Edit Online

Click **Show Version** on the **View** menu to display version information about the debugger and all loaded extension DLLs. This information is displayed in the Debugger Command window.

This command is equivalent to pressing CTRL+ALT+W (and pressing CTRL+W in KD).

This command has the same effect as the **version (Show Debugger Version)** command, except that the latter command also displays the version of the Microsoft Windows operating system.

View | Toolbar

12/5/2018 • 2 minutes to read • Edit Online

Select or clear **Toolbar** on the **View** menu to cause the toolbar to appear or disappear.

For more information about how to use the toolbar, see Using the Toolbar and Status Bar. For more information about each toolbar button, see Toolbar Buttons.

View | Status Bar

12/5/2018 • 2 minutes to read • Edit Online

Select or clear **Status Bar** on the **View** menu to cause the status bar to become visible or invisible.

For more information about how to use the status bar, see Using the Toolbar and Status Bar.

View | Font

12/5/2018 • 2 minutes to read • Edit Online

Click **Font** on the **View** menu to change the font that appears in the debugging information windows.

This command is equivalent to clicking the **Font** button ( ) on the toolbar.

Dialog Box

When you click **Font**, the **Font** dialog box appears. In this dialog box, you can select the font, style, and size from the appropriate lists,. You can also select the script from a drop-down menu to get the appropriate alphabet. To accept your changes, click **OK**.

Click **Cancel** to cancel changes to the font.

Additional Information

For more information about how to change the character display of the debugging information windows, see Changing Text Properties.

View | Options

12/5/2018 • 3 minutes to read • Edit Online

Click **Options** on the **View** menu to open the **Options** dialog box. This command is equivalent to clicking the button ( ) on the toolbar.

Dialog Box

In the **Options** dialog box, you can select or deselect the following options:

Tab width

The **Tab width** box controls how tab characters are displayed in any Source window. In the **Tab width** box, enter the number of spaces that you want to have between each tab setting. (The default setting is 8. For more information about text properties, see Changing Text Properties.)

Reuse after opening this many

The **Reuse after opening this many** box controls the number of document, or source, windows that can be open at the same time. If the specified number of source windows has been reached, opening a new window causes an existing window to close. Windows that are marked as tab-dock targets do not close. The last windows to close are the ones you used most recently.

Parse source languages

If the **Parse source languages** check box is selected, the text of source code in all Source windows is colored according to a simple parse of the source syntax. To change the colors, in the **Colors** area of the dialog box, select a syntax element and then click **Change**. (To turn the syntax colors off in a single Source window, open that window's shortcut menu, click **Select source language**, and then click **<None>**.)

Evaluate on hover

If the **Evaluate on hover** check box is selected (and the **Parse source languages** check box is selected as well), symbols in a Source window will be evaluated when you select that window and then hover over a symbol with the mouse. The evaluation is the same as that produced by the **dt (Display Type)** command.

Enter repeats last command

If the **Enter repeats last command** check box is selected, you can press the ENTER key at an empty prompt in the Debugger Command window to repeat the previous command. If you clear this check box, the ENTER key generates a new prompt.

Automatically scroll

The **Automatically scroll** check box controls the automatic scrolling that occurs when new text is sent to the Debugger Command window. If you want to turn off this feature, clear the **Automatically scroll** check box. For more information about this scrolling, see Using Debugger Commands.

Workspace Prompts

In the **Workspace Prompts** area, you can click one of three options to determine when and how frequently the workspace is saved in WinDbg.

- If you click **Always ask**, when a workspace changes (such as when a debugging session ends), the debugger displays the **Workspace save** dialog box where you can save the workspace.

  In the **Workspace save** dialog box, if you click **Don't ask again**, WinDbg resets the **Workspace Prompts** option to **Never save** or **Always save**.

- If you click **Always save**, the workspace is saved automatically whenever it changes.

- If you click **Never save**, the workspace is not saved when it changes, and you are not prompted to save it.

QuickEdit Mode
If the **QuickEdit Mode** check box is selected, you can right-click an item to copy or paste, depending on the window selection state. When you clear this check, QuickEdit is disabled and you can right-click an item to open a shortcut menu for the window. You cannot give individual windows different settings; the QuickEdit setting applies globally to all windows. By default, this box is selected. The QuickEdit setting is saved in the current workspace.

Colors
To change the color of the source text that is displayed, select an item from the **Colors** area and then click **Change**. Select a color, or select a custom color, and then click**OK**.

In the **Colors** menu, you can change the colors of the following items:

- The first ten items represent text in the Disassembly window and the Source window.
- The **Changed data text** item represents data entries that have been changed (for example, in the Registers window).
- The ten **Source** *Xxx* items control the colors that are used for syntax elements in the Source window.
- The remaining items refer to different kinds of text in the Debugger Command window.

These color changes take effect when you click **OK**. To discard these changes, click **Cancel**.

View | Source language file extensions

12/5/2018 • 2 minutes to read • Edit Online

Click **Source language file extensions** on the **View** menu to control the file name extensions that WinDbg recognizes as source file name extensions. You can also specify which programming languages are associated with which file name extensions.

Dialog Box

When you click **Source language file extensions**, the **File Extensions for Source Languages** dialog box appears. In this dialog box, you can add or delete file name extensions by inserting the cursor in the **Extensions and languages** box and typing the appropriate information. Make sure that you specify the appropriate programming language for each file name extension. For example, **cxx=C++** indicates that a .cxx file name extension is a source file and that the corresponding programming language of any file that has that extension is C++. Click **OK** to implement your changes, or click **Cancel** to discard any changes.

View | WinDbg Command Line

12/5/2018 • 2 minutes to read • Edit Online

Click **WinDbg Command Line** on the **View** menu to display the command that was used to open the current WinDbg session.

The command appears in a small message window. Click **OK** to close this window.

Debug Menu

12/5/2018 • 2 minutes to read • Edit Online

This section describes the following commands on the **Debug** menu of WinDbg:

Debug | Go

Debug | Go Unhandled Exception

Debug | Go Handled Exception

Debug | Restart

Debug | Stop Debugging

Debug | Detach Debuggee

Debug | Break

Debug | Step Into

Debug | Step Over

Debug | Step Out

Debug | Run to Cursor

Debug | Source Mode

Debug | Resolve Unqualified Symbols

Debug | Event Filters

Debug | Modules

Debug | Kernel Connection | Cycle Baud Rate

Debug | Kernel Connection | Cycle Initial Break

Debug | Kernel Connection | Resynchronize

Debug | Go

12/5/2018 • 2 minutes to read • Edit Online

Click **Go** on the **Debug** menu to resume (or begin) execution on the target. This execution will continue until a breakpoint is reached, an exception or event occurs, the process ends, or the debugger breaks into the target.

This command is equivalent to pressing F5 or clicking the **Go (F5)** button ( ) on the toolbar.

Additional Information

For more information about the effects of this action, other methods of issuing this command, and other ways to control program execution, see Controlling the Target.

Debug | Go Unhandled Exception

12/5/2018 • 2 minutes to read • Edit Online

Click **Go Unhandled Exception** on the **Debug** menu to resume execution on the target and to treat the current exception as unhandled.

Additional Information

For more information about the effects of this action, other methods of issuing this command, and other ways to control program execution, see Controlling the Target. For more information about exceptions and other events, see Controlling Exceptions and Events.

Debug | Go Handled Exception

12/5/2018 • 2 minutes to read • Edit Online

Click **Go Handled Exception** on the **Debug** menu to resume execution on the target and to treat the current exception as handled.

Additional Information

For more information about the effects of this action, other methods of issuing this command, and other ways to control program execution, see Controlling the Target. For more information about exceptions and other events, see Controlling Exceptions and Events.

Debug | Restart

12/5/2018 • 2 minutes to read • Edit Online

Click **Restart** on the **Debug** menu to stop the target's execution and end the target process and all its threads. This command then restarts the target execution at the beginning of the process.

This command is equivalent to pressing CTRL+SHIFT+F5 or clicking the **Restart (Ctrl+Shift+F5)** button ( ) on the toolbar.

Additional Information

For more information about the effects of this action, other methods of issuing this command, and other ways to control program execution, see Controlling the Target. For more information about how to exit WinDbg or end the debugging session, see Ending a Debugging Session in WinDbg.

Debug | Stop Debugging

12/5/2018 • 2 minutes to read • Edit Online

Click **Stop Debugging** on the **Debug** menu to stop the target's execution and end the target process and all its threads. This action enables you to start to debug a different target application.

This command is equivalent to pressing SHIFT+F5 or clicking the **Stop debugging (Shift+F5)** button ( ) on the toolbar.

Additional Information

For more information about the effects of this action, other methods of issuing this command, and other ways to control program execution, see Controlling the Target.

Debug | Detach Debuggee

1/3/2019 • 2 minutes to read • Edit Online

Click **Detach Debuggee** on the **Debug** menu to detach from the target application and leave it running.

Detaching from the target is supported under one of the following conditions:

- You are debugging a running user-mode target.
- You are noninvasively debugging a user-mode target.

If you are debugging a live target on Windows 2000, the **Detach Debuggee** command is not available, because this version of Windows does not support detaching from a target process.

For more information about how to exit the debugger or detach from the target, see Ending a Debugging Session in WinDbg.

Debug | Break

12/5/2018 • 2 minutes to read • Edit Online

Click **Break** on the **Debug** menu to stop the target's execution and return control to the debugger.

In user mode, this command stops the process and its threads, enabling you to regain control of the debugger. In kernel mode, this command breaks into the target computer.

You can also use this command while the debugger is active. In this situation, the command will truncate long Debugger Command window displays.

The **Break** command is equivalent to pressing CTRL+BREAK or clicking the **Break (Ctrl+Break)** button ( ) on the toolbar.

ALT+DEL

You can use **ALT+DEL** to send a **Break**. **ALT+DEL** works the same as **Break (Ctrl+Break).**

User-Mode Effects

In user mode, the **Break** command causes the target application to break into the debugger. The target application stops, the debugger becomes active, and you can enter debugger commands.

If the debugger is already active, **Break** does not affect the target application. However, you can use this command to terminate a debugger command. For example, if you have requested a long display and do not want to see any more of it, **Break** will end the display and return you to the debugger command prompt.

When you are performing remote debugging with WinDbg, you can press the Break key on the host computer's keyboard. If you want to issue a break from the target computer's keyboard, use CTRL+C on an x86-based computer.

You can press the F12 key to open a command prompt when the application that is being debugged is busy. Click one of the target application's windows and press F12 on the target computer.

Kernel-Mode Effects

In kernel mode, the **Break** command causes the target computer to break into the debugger. This command locks the target computer and wakes up the debugger.

When you are debugging a system that is still running, you must press the Break key on the host keyboard to open an initial command prompt.

If the debugger is already active, **Break** does not affect the target computer. However, you can use this command to terminate a debugger command. For example, if you have requested a long display and do not want to see any more of it, **Break** will end the display and return you to the debugger command prompt.

You can also use **Break** to open a command prompt when a debugger command is generating a long display or when the target computer is busy. When you are debugging an x86-based computer, you can also press CTRL+C on the target keyboard to have the same effect.

The SYSRQ key (or pressing ALT+SYSRQ on an enhanced keyboard) is similar. This key works from the host or target keyboard on any processor. However, this key works only if you have opened the prompt by pressing CTRL+C at least one time before.

You can disable the SYSRQ key by editing the registry. In the HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\i8042prt\Parameters registry key, create a value named **BreakOnSysRq** and set it equal to DWORD 0x0. Then, restart the computer. After you have

restarted the computer, you can press the SYSRQ key on the target computer's keyboard and it will not break into the kernel debugger.

Additional Information

The corresponding key in KD and CDB is **CTRL+C**. For more information about other ways to control program execution, see Controlling the Target.

Debug | Step Into

12/5/2018 • 2 minutes to read • Edit Online

Click **Step Into** on the **Debug** menu to execute a single instruction on the target. If the instruction is a function call, the debugger steps into the function.

This command is equivalent to pressing F11 or F8 or clicking the **Step into (F11 or F8)** button ( ) on the toolbar.

Additional Information

For more information about the effects of this action, other methods of issuing this command, and other ways to control program execution, see Controlling the Target.

Debug | Step Over

12/5/2018 • 2 minutes to read • Edit Online

Click **Step Over** on the **Debug** menu to execute a single instruction on the target. If the instruction is a function call, the whole function is executed.

This command is equivalent to pressing F10 or clicking the **Step over (F10)** button ( ) on the toolbar.

Additional Information

For more information about the effects of this action, other methods of issuing this command, and other ways to control program execution, see Controlling the Target.

Debug | Step Out

12/5/2018 • 2 minutes to read • Edit Online

Click **Step Out** on the **Debug** menu to resume running on the target. This command executes the rest of the current function and breaks when the function return is completed.

This command is equivalent to pressing SHIFT+F11 or clicking the **Step out (Shift+F11)** button ( ) on the toolbar.

Additional Information

For more information about the effects of this action, other methods of issuing this command, and other ways to control program execution, see Controlling the Target.

Debug | Run to Cursor

12/5/2018 • 2 minutes to read • Edit Online

Click **Run to Cursor** on the **Debug** menu to resume running on the target. If you insert the cursor on an instruction in the Disassembly window or a Source window and then execute this action, WinDbg executes all instructions from the current instruction up to the instruction you have selected.

This command is equivalent to pressing F7 or CTRL+F10 or clicking the **Run to cursor (Ctrl+F10 or F7)** button ( ) on the toolbar.

Additional Information

For more information about the effects of this action, other methods of issuing this command, and other ways to control program execution, see Controlling the Target.

Debug | Source Mode

12/5/2018 • 2 minutes to read • Edit Online

Select **Source Mode** on the **Debug** menu to switch the debugger to source mode. Clear **Source Mode** to switch the debugger to assembly mode.

You can click the **Source mode on** button (shown on the left in the following figure) to change the debugger to source mode or click the **Source mode off** button (shown on the right in the following figure) to change the debugger to assembly mode.

When source mode is active, ASM is unavailable on the status bar. When assembly mode is active, ASM is displayed on the status bar.

Additional Information

For more information about source-mode debugging, see Debugging in Source Mode. For more information about assembly-mode debugging, see Debugging in Assembly Mode.

Debug | Resolve Unqualified Symbols

12/5/2018 • 2 minutes to read • Edit Online

Select **Resolve Unqualified Symbols** on the **Debug** menu to resolve symbols that have no module prefix.

If you clear **Resolve Unqualified Symbols**, the debugger cannot resolve symbols that have no module prefix. If you do not select **Resolve Unqualified Symbols** and a variable that has no prefix is not already loaded, the debugger does not load any additional symbols to resolve it. You can still use unqualified symbols when this option is clear, but only if they have been previously loaded.

Although we always recommend that you use module qualifiers, you can clear the **Resolve Unqualified Symbols** option to avoid loading symbols that resolve incorrect or misspelled symbols when module qualifiers are not used.

Additional Information

For more information about symbols, loading symbols, and verifying symbols, see Symbols.

Debug | Event Filters

12/5/2018 • 2 minutes to read • Edit Online

Click **Event Filters** on the **Debug** menu to open the **Event Filters** dialog box. In this dialog box, you can configure the break status and handling status of exceptions and events.

Dialog Box

The **Event Filters** dialog box lists all events that the debugger recognizes. You may add numbered exceptions to the list that will then be displayed.

To change the break status for an event, select the event and then click one of the **Execution** option buttons (**Enabled**, **Disabled**, **Output**, or **Ignore**).

To change the handling status for an event, select the event and then click one of the **Continue** option buttons (**Handled** or **Not Handled**).

To add a new numbered exception, click **Add**. When the **Exception Filter** dialog box appears, enter the exception code, click the appropriate button for the break status and handling status, and then click **OK**.

To remove a numbered exception, select the exception and then click **Remove**. You cannot remove the standard events.

When you set the status for the **Load module** or **Unload module** events, you can limit this status to a specific module. Click **Argument**, enter the name of the module or the base address of the module in the **Filter Argument** dialog box, and then click **OK**. You can use wildcards when you specify the base address. If you do not specify a module, the break occurs when any module is loaded or unloaded.

When you set the status for the **Debuggee output** event, you can limit this status to a specific output pattern. Click **Argument**, enter the output pattern in the **Filter Argument** dialog box, and then click **OK**. If you do not specify an output pattern, the break occurs for any output.

If you want to set automatic commands that are executed if the event breaks into the debugger, select the event and then click **Commands**. The **Filter Command** dialog box will appear. Enter any commands that you want into the **Command** or **Second-chance Command** box. Separate multiple commands by using semicolons and do not enclose these commands in quotation marks.

Additional Information

For more information about break status and handling status, all event codes, the default status for all events, and other methods of controlling this status, see Controlling Exceptions and Events.

Debug | Modules

12/5/2018 • 2 minutes to read • Edit Online

Click **Modules** on the **Debug** menu to display the current list of loaded modules.

Dialog Box

When you click **Modules**, the **Module List** dialog box appears. This dialog box lists all modules that are currently loaded into memory.

Module List is divided into the following columns:

- The **Name** column specifies the module name.
- The **Start** and **End** columns specify the first and last address of the module's memory image.
- The **Timestamp** column specifies the build date and time for the module.
- The **Checksum** column specifies the checksum value.
- The **Symbols** column displays information about the symbols that this module uses. For more information about the values that appear in this column, see Symbol Status Abbreviations.
- The **Symbol file** column specifies the path and file name of the associated symbol file. If the debugger is unaware of any symbol file, the name of the executable file is given instead.

If you click the title bar of a column, the display is sorted by the data in that column. If you click the title bar again, the sort order reverses.

If you select a line and then click **Reload**, that module's symbol information is reloaded.

If you select a line and press CTRL+C, the whole line is copied to the clipboard.

Click **Close** to close this dialog box.

Debug | Kernel Connection | Cycle Baud Rate

12/5/2018 • 2 minutes to read • Edit Online

Point to **Kernel Connection** and then click **Cycle Baud Rate** on the **Debug** menu to change the baud rate that is used in the kernel debugging connection.

This command is equivalent to pressing CTRL+ALT+A. (You can also press CTRL+A in KD.)

This command cycles through all available baud rates for the kernel debugging connection. Supported baud rates are 19200, 38400, 57600, and 115200. Every time that you use this command, the next baud rate is selected. If the baud rate is already at 115200, it is reduced to 19200.

You cannot use this command to change the baud rate at which you are debugging. The baud rate of the host computer and the target computer must match, and you cannot change the baud rate of the target computer without restarting the computer. Therefore, you must change the baud rate only if the two computers are trying to communicate at different rates. In this case, you must change the host computer's baud rate to match that of the target computer.

Debug | Kernel Connection | Cycle Initial Break

12/5/2018 • 2 minutes to read • Edit Online

Point to **Kernel Conection** and then click **Cycle Initial Break** on the **Debug** menu to change the conditions on which the debugger automatically breaks into the target computer.

This command is equivalent to pressing CTRL+ALT+K. (You can also press CTRL+K in KD.)

This command causes the kernel debugger to cycle through the following three states:

No break
The debugger does not break into the target computer unless you press CTRL+BREAK or Debug | Break.

Break on reboot
The debugger breaks into a restarted target computer after the kernel initializes. This command is equivalent to starting WinDbg with the -b **command-line option**.

Break on first module load
The debugger breaks into a restarted target computer after the first kernel module is loaded. (This action causes the break to occur earlier than in the **Break on reboot** state.) This command is equivalent to starting WinDbg with the -d **command-line option**.

When you use the **Cycle Initial Break** command, the new break state is displayed.

Additional Information

For more information about related commands and an explanation of how the restart process affects the debugger, see Crashing and Rebooting the Target Computer.

Debug | Kernel Connection | Resynchronize

12/5/2018 • 2 minutes to read • Edit Online

Point to **Kernel Connection** and then click **Resynchronize** on the **Debug** menu to cause the debugger to try to reestablish a kernel debugging connection with the target computer.

This command is equivalent to pressing CTRL+ALT+R. (You can also press CTRL+R in KD.)

Use this command if the target is not responding.

Additional Information

For more information about reestablishing contact with the target, see Synchronizing with the Target Computer.

Window Menu

12/5/2018 • 2 minutes to read • Edit Online

This section describes the following commands on the **Window** menu of WinDbg:

Window | Close All Source Windows

Window | Close All Error Windows

Window | Open Dock

Window | Dock All

Window | Undock All

Window | Cascade Floating Windows

Window | Horizontally Tile Floating Windows

Window | Vertically Tile Floating Windows

Window | MDI Emulation

Window | Automatically Open Disassembly

List of Open Windows

Window | Close All Source Windows

12/5/2018 • 2 minutes to read • Edit Online

Click **Close All Source Windows** on the **Window** menu to close all Source windows.

Window | Close All Error Windows

12/5/2018 • 2 minutes to read • Edit Online

Click **Close All Error Windows** on the **Window** menu to close all error message boxes that have opened from source files that were not found.

Window | Open Dock

12/5/2018 • 2 minutes to read • Edit Online

Click **Open Dock** on the **Window** menu to create a new dock. A *dock* is an independent window that you can drag debugging information windows to. For more information about docks, see Creating New Docks.

Additional Information

For more information about docked, tabbed, and floating windows, see Positioning the Windows.

Window | Dock All

12/5/2018 • 2 minutes to read • Edit Online

Click **Dock All** on the **Window** menu to dock all of the floating windows, except for those with **Always floating** selected on their shortcut menus.

WinDbg automatically positions each floating window. If a window has never been docked before, WinDbg moves it to a new untabbed location. If a window has been docked before, WinDbg moves it to its most recent docking location, which might be either tabbed or untabbed.

Additional Information

For more information about docked, tabbed, and floating windows, see Positioning the Windows.

Window | Undock All

12/5/2018 • 2 minutes to read • Edit Online

Click **Undock All** on the **Window** menu to change all of the docked windows to floating windows.

WinDbg returns each docked window to the position that it occupied the last time that it was a floating window.

Additional Information

For more information about docked, tabbed, and floating windows, see Positioning the Windows.

Window | MDI Emulation

12/5/2018 • 2 minutes to read • Edit Online

Select **MDI Emulation** on the **Window** menu to cause WinDbg to emulate the Multiple Document Interface (MDI) style of windowing. This kind of windowing differs from docking mode because all windows are floating, but floating windows are constrained within the frame window. This behavior emulates the behavior of WinDbg before docking mode was introduced.

Clear **MDI Emulation** on the **Window** menu to return WinDbg to docking mode.

Additional Information

For more information about docked, tabbed, and floating windows, see Positioning the Windows.

Window | Automatically Open Disassembly

12/5/2018 • 2 minutes to read • Edit Online

Select **Automatically Open Disassembly** on the **Window** menu to cause the Disassembly window to open every time WinDbg begins a debugging session.

If you clear this command, you can still open the Disassembly window by clicking **Disassembly** on the **View** menu, pressing ALT+7, or clicking the **Disassembly (Alt+F7)** button ( ) on the toolbar.

List of Open Windows

12/5/2018 • 2 minutes to read • Edit Online

At the bottom of the **Window** menu, there is a list of all currently open debugging information windows. Click any window from this list to open the window.

Help Menu

12/5/2018 • 2 minutes to read • Edit Online

This section describes the following commands on the **Help** menu of WinDbg:

Help | Contents

Help | Window

Help | Selection

Help | Index

Help | Search

Help | About

Help | Contents

12/5/2018 • 2 minutes to read • Edit Online

Click **Contents** on the **Help** menu to open the **Contents** tab in this Help documentation.

This command is equivalent to pressing F1.

Additional Information

For more information about how to use this Help file, see Using the Help File.

Help | Index

12/5/2018 • 2 minutes to read • Edit Online

Click **Index** on the **Help** menu to open the **Index** tab in this Help documentation.

Additional Information

For more information about how to use this Help file, see Using the Help File.

Help | Search

12/5/2018 • 2 minutes to read • Edit Online

Click **Search** on the **Help** menu to open the **Search** tab in this Help documentation.

Additional Information

For more information about how to use this Help file, see Using the Help File.

Help | About

12/5/2018 • 2 minutes to read • Edit Online

Click **About** on the **Help** menu to open a message box that shows the version information for the WinDbg binaries that you are using.

Click **OK** to close this message box.

Toolbar Buttons

12/5/2018 • 2 minutes to read • Edit Online

Except for the breakpoint button, each button on the toolbar is equivalent to a menu command. For a full description of each button's effects, see the page for the corresponding menu command.

The buttons on the toolbar have the following effects.

BUTTON	DESCRIPTION
	Opens a source file as a read-only file. Equivalent to File | Open Source File.
	Removes the selected text from the active window and puts it on the clipboard. Equivalent to Edit | Cut.
	Copies the selected text from the active window to the clipboard. Equivalent to Edit | Copy.
	Pastes the text on the clipboard to where the cursor is located. Equivalent to Edit | Paste.
	Starts or resumes execution. Execution continues until a breakpoint is reached, an exception or event occurs, the process ends, or the debugger breaks into the target. Equivalent to Debug | Go.
	Restarts execution at the beginning of the process. Equivalent to Debug | Restart.
	Stops execution and terminates the target process permanently. Equivalent to Debug | Stop Debugging.
	In user mode, this button stops the process and its threads. In kernel mode, this button breaks into the target computer. Control is returned to the debugger. This button is also useful for cutting off long Debugger Command window displays. Equivalent to Debug | Break.
	Executes a single instruction. If the instruction is a function call, the debugger steps into the function. Equivalent to Debug | Step Into.
	Executes a single instruction. If the instruction is a function call, the debugger executes the whole function in one step. Equivalent to Debug | Step Over.

BUTTON	DESCRIPTION
	Executes the rest of the current function, and breaks when the function return is completed. Equivalent to Debug | Step Out.
	Executes all instructions from the current instruction up to the instruction marked in the active Disassembly window or Source window. Equivalent to Debug | Run to Cursor.
	If the active window is a Source or Disassembly window: Inserts a breakpoint at the current line. (If there already is a breakpoint set at the current line, this button removes the breakpoint.) **Otherwise:** Opens the **Breakpoints** dialog box like Edit | Breakpoints.
	Opens or activates the Debugger Command window. Equivalent to View | Command.
	Opens or activates the Watch window. Equivalent to View | Watch.
	Opens or activates the Locals window. Equivalent to View | Locals.
	Opens or activates the Registers window. Equivalent to View | Registers.
	Opens a new Memory window. Equivalent to View | Memory.
	Opens or activates the Calls window. Equivalent to View | Call Stack.
	Opens or activates the Disassembly window. Equivalent to View | Disassembly.
	Opens or activates the Scratch Pad. Equivalent to View | Scratch Pad.
	Switches between source-mode and assembly-mode debugging. Equivalent to selecting or clearing Debug | Source Mode.

BUTTON	DESCRIPTION
	Enables you to change the font that is used in the debugging information windows. Equivalent to View | Font.
	Displays the **Options** dialog box. Equivalent to View | Options.

Keyboard Shortcuts

12/5/2018 • 4 minutes to read • Edit Online

You can use the following keyboard shortcuts to switch between windows. For more information about how to move between the windows, see Positioning the Windows.

KEYS	EFFECT
CTRL+TAB	Switches between debugging information windows. By using this key repeatedly, you can scan through all of the windows, regardless of whether they are floating, docked by themselves, or part of a tabbed collection of docked windows.
ALT+TAB	Switches between the windows that are currently on your desktop. You can also use this keyboard shortcut to switch between the WinDbg frame and any additional docks you have created.

You can use the following keyboard shortcuts instead of the mouse to select menu commands. For more information about each command, see the individual command topics.

KEYS	MENU EQUIVALENT
F1	Help | Contents
F3	Edit | Find Next
SHIFT+F3	Same as Edit | Find Next, except the search is performed in the reverse direction.
ALT+F4	File | Exit
CTRL+F4	File | Close Current Window
F5	Debug | Go
SHIFT+F5	Debug | Stop Debugging
CTRL+SHIFT+F5	Debug | Restart
F6	File | Attach to a Process

KEYS	MENU EQUIVALENT
F7	Debug | Run to Cursor
F8	Debug | Step Into
F9	If the active window is a Source or Disassembly window: Inserts a breakpoint at the current line. (If there already is a breakpoint set at the current line, this button removes the breakpoint.) Otherwise: Opens the **Breakpoints** dialog box like Edit | Breakpoints.
ALT+F9	Edit | Breakpoints
F10	Debug | Step Over
CTRL+F10	Debug | Run to Cursor
F11	Debug | Step Into
SHIFT+F11	Debug | Step Out
ALT+1	Opens the Debugger Command window (same as View | Command).
ALT+SHIFT+1	Closes the Command window.
ALT+2	Opens the Watch window (same as View | Watch).
ALT+SHIFT+2	Closes the Watch window
ALT+3	Opens the Locals window (same as View | Locals)
ALT+SHIFT+3	Closes the Locals window.
ALT+4	Opens the Registers window (same as View | Registers).
ALT+SHIFT+4	Closes the Registers window.
ALT+5	Opens a new Memory window (same as View | Memory).

KEYS	MENU EQUIVALENT
ALT+SHIFT+5	Closes the Memory window.
ALT+6	Opens the Calls window (same as View | Call Stack).
ALT+SHIFT+6	Closes the Calls window
ALT+7	Opens the Disassembly window (same as View | Disassembly).
ALT+SHIFT+7	Closes the Disassembly window.
ALT+8	Opens the Scratch Pad (same as View | Scratch Pad).
ALT+SHIFT+8	Closes the Scratch Pad.
ALT+9	Opens the Processes and Threads window (same as View | Processes and Threads).
ALT+SHIFT+9	Closes the Processes and Threads window.
CTRL+A	Edit | Select All
CTRL+C	Edit | Copy
CTRL+D	File | Open Crash Dump
CTRL+E	File | Open Executable
CTRL+F	Edit | Find
CTRL+G	Edit | Go to Address
CTRL+I	File | Image File Path
CTRL+SHIFT+I	Edit | Set Current Instruction
CTRL+K	File | Kernel Debug

KEYS	MENU EQUIVALENT
CTRL+L	Edit | Go to Line
CTRL+O	File | Open Source File
CTRL+P	File | Source File Path
CTRL+R	File | Connect to Remote Session
CTRL+S	File | Symbol File Path
CTRL+V	Edit | Paste
CTRL+SHIFT+V	Edit | Evaluate Selection
CTRL+W	File | Open Workspace
CTRL+X	Edit | Cut
CTRL+SHIFT+Y	Edit | Display Selected Type
ALT+* (number keypad)	Edit | Go to Current Instruction
SHIFT+DELETE	Edit | Cut
SHIFT+INSERT	Edit | Paste
CTRL+INSERT	Edit | Copy
CTRL+BREAK	Debug | Break
ALT+DEL	Debug | Break

The following shortcut keys are equivalent to KD / CDB control keys.

KEYS	MENU EQUIVALENT	KD / CDB CONTROL KEY
CTRL+ALT+A	Debug | Kernel Connection | Cycle Baud Rate	CTRL+A

KEYS	MENU EQUIVALENT	KD / CDB CONTROL KEY
CTRL+ALT+D		**CTRL+D (Toggle Debug Info)**
CTRL+ALT+K	Debug | Kernel Connection | Cycle Initial Break	CTRL+K
CTRL+ALT+R	Debug | Kernel Connection | Resynchronize	CTRL+R
CTRL+ALT+V	View | Verbose Output	CTRL+V
CTRL+ALT+W	View | Show Version	CTRL+W

You can use the following keyboard shortcuts to move the caret (^) in most of the debugging information windows.

CARET MOVEMENT	KEY
One character left	LEFT
One character right	RIGHT
Word left	CTRL+LEFT
Word right	CTRL+RIGHT
Line up	UP
Line down	DOWN
Page up	PAGE UP
Page down	PAGE DOWN
Beginning of the current line	HOME
End of the line	END
Beginning of the file	CTRL+HOME

CARET MOVEMENT	KEY
End of the file	CTRL+END

Note In the Debugger Command window, the UP and DOWN keys browse through the command history. You can use the INSERT key to turn insert mode on and off.

Use the following keyboard shortcuts to select text.

SELECT	KEYS
Character to the left	SHIFT+LEFT
Character to the right	SHIFT+RIGHT
Word to the left	SHIFT+CTRL+LEFT
Word to the right	SHIFT+CTRL+RIGHT
Current line	SHIFT+DOWN if the caret is in column 1
Line above	SHIFT+UP if the caret is in column 1
To the end of the line	SHIFT+END
To the beginning of the line	SHIFT+HOME
Screen up	SHIFT+PAGE UP
Screen down	SHIFT+PAGE DOWN
To beginning of file	SHIFT+CTRL+HOME
To end of file	SHIFT+CTRL+END

Use the following keyboard shortcuts to delete text.

DELETE	KEY
Character to the right of caret	DELETE
Character to the left of caret	BACKSPACE

DELETE	KEY
Selected text	DELETE