

The NT Insider

A publication of OSR Open Systems Resources, Inc.



MSFT to Non-Win10 Users—No More Driver Updates for You!

I don't know if you've heard the news yet, but in case you haven't: Microsoft has plans to make it impossible to update drivers on Win 7, Win 8.1 and Windows Server 2012 R2 – including Windows Embedded 8 and Windows Embedded 8.1. You can [read the policy here](#). You can read a more complete explanation and analysis of this issue in [my blog post on this topic from Mid-October, here](#).

Most people who hear about this think I'm exaggerating. Or that I've misread the doc pages. But, I assure you, after talking with my friends up in Redmond they have confirmed to me that in fact this is the plan: Starting early next year it *will no longer be possible to cross-sign new drivers*. The only 8.1 or Windows Server 2012 R2 will be WHQL/WUK Certified

We had heard... or maybe just hoped... that Microsoft was going to extend Attestation Signing to versions of Windows other than Windows 10. Sadly, that has not materialized.

Making VS Static Analysis Tools Work For You (Not Annoy You)

My C programming, and especially my attempts at using C++ “as a better C”, can usually use a lot of help. I admit it. My code quality can be inconsistent. Sometimes it’s because I’m not fully dedicated to a specific coding convention to use: Do I want to continue using `NULL` or should I switch to `nulptr`? Sometimes it’s because I’m initially focused on getting my device to work, and I take some messy shortcuts. That’s fine, except when those shortcuts get accidentally left in the code at Final Release. Stuff like this:

```

// Set the BME280 into
// temp and pressure.
//
writeBuffer[0] = BME280
writeBuffer[1] = 0xB7;
status = Bme280WriteData

```

6,523 kernel developers are waiting to read
The NT Insider

Admit

OK, that's not the worst code ever written, but it's not "both temp and pressure" but should be "value".

I'm also sort of famous for coding or cut/pasting a call to some function and then neglecting to check the return status. Yeah, I know: sloppy. I'm not proud of it. Sometimes my mind just starts working faster than my fingers.

Five C++ Features Even a C Programmer Will Love

I have long held that C++ sucks, and that it sucks really, really, badly. I have been so very vocal about this over the past 20 or so years, that almost anyone who's been involved in Windows driver development and has read [NTDEV](#) even a little has seen me hold forth on this topic.

I am not alone in this belief. In fact, no lesser beings than Mr. Penguin Pants and Elon Musk agree with me. Ken Thompson and Rob Pike apparently also agree.

```
From: Linus Torvalds <torvalds@linux-foundation.org>  
Subject: Re: [RFC] Convert builtin-mallinfo() to use the Better String Library.  
To: "Hengrui Shao" <shao.hengrui@gmail.com>  
Message-ID: <2007-09-06T17:58:08Z> (2 years, 4 weeks, 16 hours and 36 minutes ago)  
Date: 2007-09-06T17:58:08Z  
  
> On Wed, 5 Sep 2007, Dmitry Kalinur wrote:  
>  
> I have first looked at file sources code that things struggle me as well.  
> I think C++ is more portable than C, because it has less platform portability  
> issues. C++ is more portable than C, because it has less platform portability
```

"C++ is a horrible language. It's made more horrible by the fact that a lot of people who are supposed to be experts in the language are just as eager to generate trash and utter crap with it. (Quite frankly, even if the choice of C++ were to do "nothing" but keep the C++ programmers out, that in itself would be a huge reason to use C++.)"

A Generic Device Class Filter Using WDF

How many times have you been working on a device or a driver project, seen some behavior in one of the existing Windows device stacks, and asked yourself “I wonder what’s going on with that particular sequence of I/O requests” or “What Control Codes, exactly, does that driver handle”?

Given that we don't have the source code to most of the drivers that ship with Windows, getting these answers might seem daunting. But, fortunately, in Windows we have Filter Drivers.

In this article, we're going to provide a brief overview of device Class Filters and present a tutorial walk-through of an example of a generic device Class Filter. As a part of this tutorial, we'll describe the primary differences between WDF function drivers and WDF filter drivers, and how you receive, examine, and send requests to the next device in the device stack.

The source code for the example driver we'll be discussing is available on GitHub.

Let's start with some general background on Windows filter drivers.

Different Types of Filter Drivers

As you probably already know, a Filter Driver is a uniquely powerful Windows component that allows you to insert a "Filter Device Object" above or below a given Device Object in a Windows device stack. This allows you to provide a driver that observes, manages, or even modifies I/O operations as they are processed. If you're having trouble remembering your Windows device stack, you can find out more about it in the next chapter.

Published by

OSR Open Systems Resources, Inc.
889 Elm Street, 6th Floor
Manchester, New Hampshire 03101 USA
(v) +1.603.595.6500
(f) +1.603.595.6503

<http://www.osr.com>

Consulting Partner

Peter G. Viscarola

Engineering Partner

Scott Noone

Executive Editor

Daniel D. Root

Contributing Editors

OSR Staff

Send Stuff To Us:

NTInsider@osr.com

Single Issue Price: \$15.00

The NT Insider is Copyright ©2020 All rights reserved. No part of this work may be reproduced or used in any form or by any means without the written permission of OSR Open Systems Resources, Inc.

We welcome both comments and unsolicited manuscripts from our readers. We reserve the right to edit anything submitted, and publish it at our exclusive option.

Stuff Our Lawyers Make Us Say

All trademarks mentioned in this publication are the property of their respective owners. "OSR", "OSR Online" and the OSR corporate logo are trademarks or registered trademarks of OSR Open Systems Resources, Inc.

We really try very hard to be sure that the information we publish in *The NT Insider* is accurate. Sometimes we may screw up. We'll appreciate it if you call this to our attention, if you do it gently.

OSR expressly disclaims any warranty for the material presented herein. This material is presented "as is" without warranty of any kind, either expressed or implied, including, without limitation, the implied warranties of merchantability or fitness for a particular purpose. The entire risk arising from the use of this material remains with you. OSR's entire liability and your exclusive remedy shall not exceed the price paid for this material. In no event shall OSR or its suppliers be liable for any damages whatsoever.

It is the official policy of OSR Open Systems Resources, Inc. to safeguard and protect as its own, the confidential and proprietary information of its clients, partners, and others. OSR will not knowingly divulge trade secret or proprietary information of any party without prior written permission. All information contained in *The NT Insider* has been learned or deduced from public sources...often using a lot of sweat and sometimes even a good deal of ingenuity.

OSR is fortunate to have customer and partner relations that include many of the world's leading high-tech organizations. As a result, OSR may have a material connection with organizations whose products or services are discussed, reviewed, or endorsed in *The NT Insider*.

Neither OSR nor *The NT Insider* is in any way endorsed by Microsoft Corporation. And we like it that way, thank you very much.

Inside This Issue

Corona-Quarantine Got You Down?

[Skip the Travel...Attend OSR Seminars Remotely.....](#)3

Peter Pontificates:

[Microsoft to Non-Win10 Users: No Driver Updates for You!.....](#)4

21st Century Driver Writing

[Five C++ Features Even a C Programmer Will Love.....](#)6

New Options for Compile-Time Validation

[Making VS Static Analysis Tools Work For You \(Not Annoy You\).....](#)13

Learning By Example

[A Generic Device Class Filter Using WDF.....](#)21

Get Social with OSR Real-Time Updates

Just in case you're not already following us on Twitter, Facebook or LinkedIn, below are a few of the more recent contributions that are getting attention in the Windows driver development community:

Microsoft: No Driver Updates Allowed for Win7 and Win8

Microsoft has announced that it is ending the ability to cross-sign drivers, effective 1 July 2021.

This will effectively make it impossible to release new or updated drivers for Windows

<https://www.osr.com/blog/2020/10/15/microsoft-driver-updates-allowed-win7-win8/>

Bug in New Function ExAllocatePoolZero Results in Security Vulnerability and Crashes

OSR found and reported a bug to Microsoft that has both security and reliability implications for driver developers.

<https://www.osr.com/blog/2020/07/14/bug-in-new-function-exallocatepoolzero-results-in-security-vulnerability-and-crashes/>

NTSTATUS to Win32 Error Code Mappings

Some time ago, for reasons known only to our friends in Redmond, the Microsoft Knowledge Base article that listed all the NTSTATUS values and their equivalent Win32 ERROR mappings disappeared...

<https://www.osr.com/blog/2020/04/23/ntstatus-to-win32-error-code-mappings/>

Beware: VS 2019 V16.4.x Update Breaks the WDK

(Update: issue resolved) If you're like most people, when Visual Studio lights the little "there's an update available" icon, you get the update installed as soon as convenient. After all, these updates (which seem to come out every other week or so) often fix real (annoying) problems in Visual Studio. So, most of us figure, better to get them installed and see what gets fixed...but...

<https://www.osr.com/blog/2020/03/31/beware-vs-2019-v16-4-x-update-breaks-the-wdk/>

Finally! Attend OSR's Driver Development Seminars Online

One of the most common requests that we've received over the years about our seminars has been to allow people to attend remotely, online, via the Internet.

<https://www.osr.com/blog/2019/09/10/finally-attend-osrs-driver-development-seminars-online/>

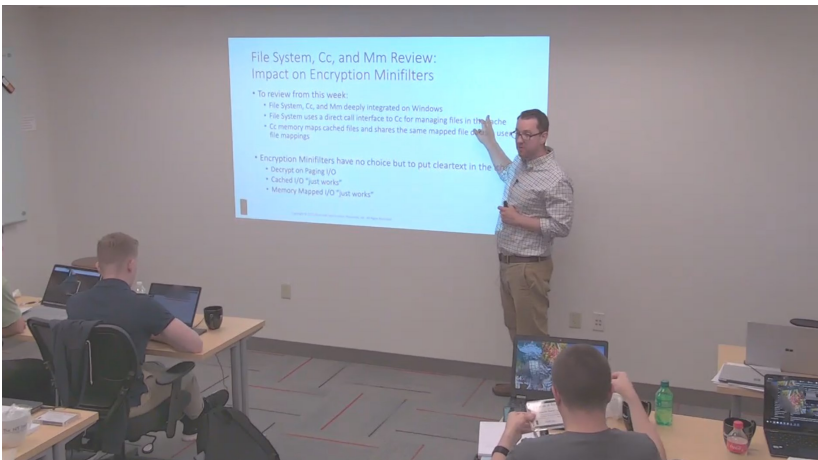


Follow us!



Corona-Quarantine Got You Down?

Skip the Travel... Attend OSR Seminars Remotely!



OSR live-streams seminars to an online audience!

You need to continue learning. And, honestly, it would be best if you could attend one of our seminars in person. But, COVID. Nobody wants to get on an airplane to attend a seminar in a conference room packed with people.

Good thing we started letting folks attend our seminars live, via the Internet, right?

Our online seminars provide you with a high-quality seminar experience. We send you hardcopies of the seminar handouts in advance of your seminar. We broadcast the seminar in HD from the OSR Seminar Space, speaking to projected slides. We listen for, and answer, live

questions over the Internet. So, when we talk about "live via the Internet" we're not talking about having one of our instructors teach your seminar from their house, sitting in front of a cheesy webcam, while their dog barks in the background. We're talking about a professional seminar experience, that's almost as good as if you were attending in-person. But, without the chance of catching The Rona.

Check it out: [Here's a sample](#) of one of our seminars that was recorded directly from a participant's Zoom session. This was the online presentation of an in-person seminar. But, even though we are no longer allowing in-person attendees, our seminar presentations are exactly the same.

Now Available Online!
Live-Streamed via the Web

Developing FS Minifilters

Hear What Our Students Have to Say!

Scott is elite. That's all I have to say.

Veni, vidi, vabooty. We came, we saw, we kicked butt.

(editor: I don't even know what that means, but this student was clearly happy!)

Next presentation:

ONLINE
8 March 2021

Peter Pontificates MSFT to Non-Win10 Users—No More Driver Updates for You!



I don't know if you've heard the news yet, but in case you haven't: Microsoft has plans to make it impossible to update drivers on Win 7, Win 8.1 and Windows Server 2012 R2 – Including Windows Embedded 8 and Windows Embedded 8.1. You can [read the policy here](#). You can read a more complete explanation and analysis of this issue in [my blog post on this topic from Mid-October, here](#).

Most people who hear about this think I'm exaggerating. Or that I've misread the doc pages. But, I assure you, after talking with my friends up in Redmond they have confirmed to me that in fact this is the plan: Starting early next year *it will no longer be possible to cross-sign new drivers*. The only

drivers you'll be able to load on any flavor of Win 7, Win 8, Win 8.1 or Windows Server 2012 R2 will be WHQL/WLK Certified drivers. Period. Full stop. End of story.

We had heard... or maybe just hoped... that Microsoft was going to extend Attestation Signing to versions of Windows other than Windows 10. Sadly, that has not materialized.

One thing that makes this plan so particularly distressing is that *Windows 8.1 and Windows Server 2012 R2 are still supported. Windows 8 Embedded and Windows 8.1 Embedded are still supported.*

So, this leaves customers who are running on any of these systems with three options, none of which are the least bit realistic or viable:

1. **Never update any drivers** – This is entirely unrealistic. There will be bugs. These bugs will need to be fixed.
2. **Get any drivers that you want to update to pass WHQL/WLK testing** – This is entirely unrealistic. There are categories of drivers, and of hardware, that will never be able to pass the WHQL tests.
3. **Upgrade to Windows 10** – This is also entirely unrealistic. People don't update well-integrated, supported, versions of the OS "just because."

I am at a loss to understand what MSFT expects the customer base to do. I am at a loss to understand how *anybody* could think this is a good plan. It feels to me like some Microsoft middle manager simply does not understand the vastness of the ways that Windows is used.

If the goal is to push people to upgrade to Win10, this is a very wrong-headed idea.

You can't just run Windows Update on a carefully configured system running Windows Embedded in a medical device, or on an aircraft, or on a system that does continuous process control in a refinery. These systems have all been specifically qualified with specific versions of Windows and updating to a different version of Windows accomplishes exactly NOTHING except destabilizing these systems.

You can't just upgrade 85,000 corporate desktops, that have been running specialized in-house LOB applications, for years.

If this policy doesn't change, I predict it will provide an impetus for folks to move OFF the Windows platform. It feels like a grossly arbitrary decision. If you can't update drivers for the version of Windows you're using, and changing to a different version of Windows would require that you requalify your entire system, maybe it's just as easy to move to a different OS entirely? Has MSFT broken trust with these users? Could this not be just one more nail in Windows' coffin in a lot of areas?

Maybe Microsoft doesn't understand, or maybe doesn't care about, the customer base that's running these systems. But we – the IHVs and ISVs that make the software that makes the world work – certainly do care.

[\(CONTINUED ON PAGE 5\)](#)

Peter Pontificates... (Cont.)

[\(CONTINUED FROM PAGE 4\)](#)

That's why we must work together to get Microsoft to change this plan. The community has done it before.

Work your technical contacts. Get the message clearly presented to the folks you work with at MSFT.

Even more important, you must explain the problem to the managers at your company who regularly engage with MSFT. You must get your execs and your managers to raise this issue with their MSFT contacts.

Because if we don't change this policy, lots of Windows users are going to be put in very difficult positions.

The time to act is now, folks. Microsoft needs to hear from you and from your management team.

The clock is ticking...



Follow us!



Peter Pontificates is a regular column by OSR Consulting Partner, Peter Viscarola. Peter doesn't care if you agree or disagree with him, but there's always the chance that your comments or rebuttal could find its way into a future issue. Send your own comments, rants or distortions of fact to: PeterPont@osr.com.

Now Available Online!
Live-Streamed via the Web

Need To Know WDF?

Tip: You can read all the articles ever published in *The NT Insider* and still not come close to what you will learn in one week in our [WDF](#) seminar. So why not join us?

Both Scott and Peter have in-depth knowledge and extensive hands-on experience writing device drivers. Their discussions about mistakes to avoid was as valuable as explaining Windows. This was the best training class that I have ever taken.

- Feedback from an attendee of THIS seminar

Seminar Outline and Information here: <http://www.osr.com/seminars/wdf-drivers/>

Next presentation:

ONLINE

7-11 December 2020

21st Century Driver Writing

Five C++ Features Even a C Programmer Will Love

I have long held that C++ sucks, and that it sucks really, really, badly. I have been so very vocal about this over the past 20 or so years, that almost anyone who's been involved in Windows driver development and has read [NTDEV](#) even a little has seen me hold forth on this topic.

I am not alone in this belief. In fact, no lesser beings than Mr. Penguin Pants and Elon Musk agree with me. Ken Thompson and Rob Pike [apparently also agree](#).

From: Linus Torvalds <torvalds@linux-foundation.org>
Subject: Re: [RFC] Convert builtin-mailinfo.c to use The Better String Library.
Newsgroups: gmane.comp.version-control.git
Date: 2007-09-06 17:50:28 GMT (2 years, 14 weeks, 16 hours and 36 minutes ago)

On Wed, 5 Sep 2007, Dmitry Kakurin wrote:

>
> When I first looked at Git source code two things struck me as odd:
> 1. Pure C as opposed to C++. No idea why. Please don't talk about portability,
> it's BS.

YOU are full of bullshit.

C++ is a horrible language. It's made more horrible by the fact that a lot of substandard programmers use it, to the point where it's much much easier to generate total and utter crap with it. Quite frankly, even if the choice of C were to do *nothing* but keep the C++ programmers out, that in itself would be a huge reason to use C.

But... I like to think that I've changed some over the years. And over time C++ has **very** much changed. Versions of C++ before C++ 11 are very different from versions of C++ since. We now have a type of C++ that's generally referred to as "Modern C++" – If you're new to this concept, there's [quite a nice description on MSDN](#).

I have, somewhat reluctantly due to the extreme level of my previous vociferousness, concluded that maybe there are *some features* of Modern C++ that aren't so bad. In fact, I think there are some features that even the most enthusiastic C Language programmer might find both useful and (more importantly) helpful in building quality into Windows drivers.

In this article, I'll describe a few of the C++ features that we

regularly use at OSR in driver development. You'll see we're not entirely "bought in" to the world of C++; But, as C++ has evolved and we've evolved, we've come a long way from our previous position which was "If you put any C++ code in one of our drivers, you will be hanged, drawn and quartered."

Limitations on C++ in Windows Drivers

Before we begin, I should probably note that there are many good and useful things in Modern C++ that cannot be used in kernel-mode. This is because in [Windows kernel-mode does not support C++ exception handling](#). C++ exception handling is entirely different from our customary kernel-mode Structured Exception Handling. As a result of this limitation, most parts of the C++ Standard Library (**std**) are off limits, including some of the most pleasant and useful features such as Containers (**std::vector**, for example) and string handling. This also means that you can't use ATL or STL – both of which are pretty much superseded by **std** anyhow. Unless, that is, you're using ATL for writing COM programs. And if you are, well... I'm sorry.

So, having mentioned some cool stuff that we *can't* use, let's have a look at five of our favorite C++ constructs that *do* work in Windows kernel-mode drivers.

#1: Strong Type Checking

Yes, I realize that this isn't a Modern C++ innovation. But it's still a massive advantage of C++ over plain old C. There can be little disagreement that strong type checking, which is inherent in C++, is a good thing. Over the past million or so years that I've been in this business, I've inherited numerous drivers that were initially written (usually by clients) in "pure" C. And I can truthfully say that I have *never*, not once, failed to find a meaningful type error when I've renamed those files with CPP extensions and rebuilt them. I think that's a good enough argument for strong type checking's value.

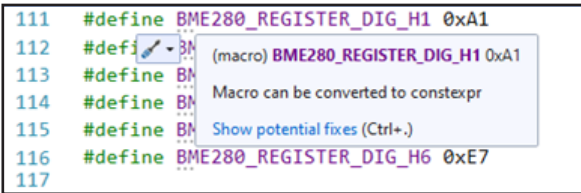
#2: constexpr instead of #define

I hope there's nobody left in the world who doesn't think that the use of preprocessor macros in C/C++ is, at best, "problematic." By way of review: Good old **#define** performs a textual substitution at compile time. This can lead to all sorts of unintentional problems, given that there's no type safety in a **#define**, arguments passed into the **#define** can get evaluated multiple times, and more. I'm not saying you should never use **#define**. I'm just saying you should use it when no other tool in your toolbox will do.

[\(CONTINUED ON PAGE 7\)](#)

C++ Features... (Cont.)

(CONTINUED FROM PAGE 6)



```

111 #define BME280_REGISTER_DIG_H1 0xA1
112 #define BME280_REGISTER_DIG_H1 0xA1
113 #define BME280_REGISTER_DIG_H2 0xE1
114 #define BME280_REGISTER_DIG_H3 0xE3
115 #define BME280_REGISTER_DIG_H4 0xE4
116 #define BME280_REGISTER_DIG_H5 0xE5
117 #define BME280_REGISTER_DIG_H6 0xE7
  
```

Macro can be converted to constexpr
Show potential fixes (Ctrl+.)

Figure 1—Visual Studio Suggests



```

111 constexpr auto BME280_REGISTER_DIG_H1 = 0xA1;
112 #define BME280_REGISTER_DIG_H2 0xE1
113 #define BME280_REGISTER_DIG_H3 0xE3
114 #define BME280_REGISTER_DIG_H4 0xE4
115 #define BME280_REGISTER_DIG_H5 0xE5
116 #define BME280_REGISTER_DIG_H6 0xE7
117
  
```

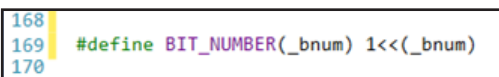
Figure 2—VS Fixes... with *auto* as the type



```

111 constexpr ULONG BME280_REGISTER_DIG_H1 = 0xA1;
112 #define BME280_REGISTER_DIG_H2 0xE1
113 #define BME280_REGISTER_DIG_H3 0xE3
114 #define BME280_REGISTER_DIG_H4 0xE4
115 #define BME280_REGISTER_DIG_H5 0xE5
116 #define BME280_REGISTER_DIG_H6 0xE7
117
  
```

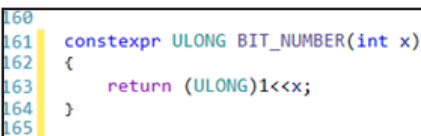
Figure 3—I'm not a fan of *auto* in this case



```

168
169 #define BIT_NUMBER(_bnum) 1<<(_bnum)
170
  
```

Figure 4—Ripe for conversion to *constexpr*



```

160
161 constexpr ULONG BIT_NUMBER(int x)
162 {
163     return (ULONG)1<<x;
164 }
165
  
```

Figure 5—Now constant at compile time

Usually **constexpr** can be used as a direct replacement for “simple” **#defines**, such as when you define bitmasks and register bit patterns. The primary difference in this case is that **constexpr** has a type. Visual Studio will even suggest that you change your basic **#defines** to **constexpr** and will offer to do the conversion for you (Figure 1).

The only disadvantage I see to letting VS have its way with your code here, is that it uses **auto** for the type... which may not be what you want (Figure 2).

I, personally, much prefer to specify the type myself. After all, I know how/when/where I'm going to use the definition (Figure 3).

You can/should also use **constexpr** in place of many of your “function like” **#define** macros. For example, the classic macro shown in Figure 4, is quite easily converted to a **constexpr**. This lets you replace the macro with the equivalent of a function, the result of which is available at compile time. Though VS won't suggest this conversion with cute little dots, it *will* do the conversion for you if you ask it to. It will go out of its way to use type **auto**, including templating the data types of any arguments – picture me rolling my eyes here. I say just assign the arguments a specific type and be done with it (Figure 5). Let's agree to delay our argument about whether and when **auto** is a good or bad thing until later in this article.

(CONTINUED ON PAGE 8)

OSR's Corporate, Online Training

Save Money, Travel Hassles; Gain Customized Expert Instruction Remotely!

We can:

- Prepare and present a one-off, private, online seminar for your team to address a specific area of deficiency or to prepare them for an upcoming project.
- Design and deliver a series of offerings with a roadmap catered to a new group of recent hires or within an existing group.
- Work with your internal training organization/HR department to offer monthly or quarterly seminars to your division or engineering departments company-wide.

Contact: Seminars@osr.com

C++ Features... (Cont.)

[\(CONTINUED FROM PAGE 7\)](#)

#3: Range-based “for” Loops

This is a feature that I don’t use every day, but when I *do* have use for it I wonder “where has this been all my life.” I love the range-based **for** loop.

Consider the following chunk-o-code, ignoring (please) whether this particular character-by-character search is wise and that it doesn’t actually *do* anything... it’s an example, in an article, after all):

```
constexpr int MAX_STRING_SIZE = 55;
WCHAR  stringBuffer[MAX_STRING_SIZE];
// Later in the code...
for(int i = 0; i < MAX_STRING_SIZE; i++) {
    if(stringBuffer[i] == L'\\') {
        break;
    }
}
```

Even this code, as shown above, is *way* better than the way I normally see such code written... which is without `MAX_STRING_SIZE` and instead with the magic literal constant value “55” used multiple times. Ugh.

Anyhow... it’s bug-prone and annoying to keep (even) the symbolic constants aligned. Assuming your storage definitions aren’t immediately adjacent to your use of that storage, you have to jump back and forth between the declaration and the **for** loop. “Was the size of **stringBuffer** `MAX_STRING_SIZE` or was it `MAX_BUFFER_SIZE` or `MAX_STRING_STORAGE`?” When you’re cranking through code, these are the kinds of things that lead to errors. We’ve all seen it.

To avoid dealing with any of this nonsense, we can use a range-based **for** loop. Like this:

```
constexpr int MAX_STRING_SIZE = 55;
WCHAR  stringBuffer[MAX_STRING_SIZE];
// Later in the code...
for (WCHAR thisChar : stringBuffer) {
    if(thisChar == L'\\') {
        break;
    }
}
```

My goodness! Isn’t that much more tidy and pleasant?? I think so. The compiler knows the size of the array **stringBuffer** and with a range-based **for**, we let the compiler deal with it for us. As a result, when you decide to change the size of the string buffer, you only have to... *change the size of the string buffer*. And the code takes care of itself. It’s not based on using the right definition. And there’s no chance of getting the definition wrong in the code.

#4: Limited Use of *auto*

I’m an extreme fan of coding with clarity, simplicity, and ease of maintenance. I dislike code that’s “clever”; I tend to over-document, sometimes to the point of “writing my C a second time in English” (which I readily admit is unwise, but I’m still in recovery from this affliction). Anyhow, my desire for clarity means that I naturally tend to avoid things like **auto** in most places where you’re supposed to be smart enough to intuit the type of a variable. So, for example, while most Modern C++ fans would probably have coded the range-based **for** above:

[\(CONTINUED ON PAGE 9\)](#)

C++ Features... (Cont.)

[\(CONTINUED FROM PAGE 8\)](#)

```
for (auto thisChar : stringBuffer) {
```

I almost always opt for being specific and clear:

```
for (WCHAR thisChar : stringBuffer) {
```

There is one case where I *love* the use of auto, however. And that's in assignments where there's a (big friggin') cast on the right hand side. For example, consider the following common case where we cast WDFCONTEXT to a Context for a WDFDEVICE in an I/O Completion Routine:

```
VOID
BasicUsbEvtRequestReadCompletionRoutine(WDFREQUEST Request,
                                         WDFIOTARGET Target,
                                         PWDF_REQUEST_COMPLETION_PARAMS Params,
                                         WDFCONTEXT Context)
{
    PBASICUSB_DEVICE_CONTEXT devContext = (PBASICUSB_DEVICE_CONTEXT)Context;
    // rest of completion routine..
```

I don't see that having the data type fully specified on the left AND the right of the combined declaration and assignment statement to be even remotely instructive. In this case, I save myself some typing by doing the following:

```
VOID
BasicUsbEvtRequestReadCompletionRoutine(WDFREQUEST Request,
                                         WDFIOTARGET Target,
                                         PWDF_REQUEST_COMPLETION_PARAMS Params,
                                         WDFCONTEXT Context)
{
    auto * devContext = (PBASICUSB_DEVICE_CONTEXT)Context;
    // rest of completion routine..
```

Notice, by the way, that I used "**auto ***" above, even though **auto** alone would work fine. Call me obsessive, but I like to be completely clear that it is a pointer that I'm declaring. Plus, clang-tidy told me I should do probably this, and who am I to argue? That's a topic we'll leave for another article, actually.

[\(CONTINUED ON PAGE 10\)](#)

Windows Internals & Software Drivers

For SW Engineers, Security Researchers & Threat Analysts

"The instructor is extremely knowledgeable regarding Windows internals. He has the communications skills to provide an informative, in-depth seminar with just the right amount of entertainment value."

- Feedback from an attendee of THIS seminar

Next Presentation:

ONLINE
25-29 January 2021

C++ Features... (Cont.)

[\(CONTINUED FROM PAGE 9\)](#)

#5: Some RAIL Patterns

Let's just ignore what the letters RAIL stand for, shall we? RAIL patterns are those that do automatic initialization when an object (such as a class) is instantiated, and automatic tear-down when the object is no longer needed. Now, C people, don't worry... I'm not going to get all OOP on you here. At least, not very much.

I tend to use RAIL patterns in my drivers when I have reasonably complex initialization I want to perform and *that initialization cannot fail* and/or when I have some specific or complex tear-down I need done when I'm done with the object.

Let me give you an example, taken from one of our projects, to illustrate where I find RAIL patterns particularly helpful. We have a driver that needs to check the access of a user that's sent us a Request. This involves acquiring the **SECURITY_SUBJECT_CONTEXT**, locking it, doing the checks, and then (before exiting the function) unlocking and releasing the **SECURITY_SUBJECT_CONTEXT**.

It's a fairly long function in which we need to do this check, and there are numerous things that can cause us to exit from the function prematurely. The code was a complete mess. That is, until I created an RAIL class that acquired and locked the **SECURITY_SUBJECT_CONTEXT** on instantiation, and then automatically unlocked and released it at function exit. You can see the code for the class in *Figure 6*.

[\(CONTINUED ON PAGE 11\)](#)

```
class OSRAuthSecurityContext
{
public:
    SECURITY_SUBJECT_CONTEXT SubjectContext{};
    PEPROCESS                Process;
    PACCESS_TOKEN            PrimaryToken;

    Explicit
    OSRAuthSecurityContext(const WDFREQUEST & Request)
    {
        PIRP irp;

        irp = WdfRequestWdmGetIrp(Request);
        Process = IoGetRequestorProcess(irp);

        if (Process == nullptr) {
            Process = IoGetCurrentProcess();
        }

        SeCaptureSubjectContextEx(nullptr,
                                   Process,
                                   &SubjectContext);

        SeLockSubjectContext(&SubjectContext);

        PrimaryToken = SubjectContext.PrimaryToken;
    }

    ~OSRAuthSecurityContext()
    {
        SeUnlockSubjectContext(&SubjectContext);
        SeReleaseSubjectContext(&SubjectContext);
    }

    //
    // Sigh... the Rule of Five requires these
    //
    OSRAuthSecurityContext(const OSRAuthSecurityContext&) = delete;           // copy constructor
    OSRAuthSecurityContext & operator=(const OSRAuthSecurityContext&) = delete; // copy assignment
    OSRAuthSecurityContext(OSRAuthSecurityContext&&) = delete;                // move constructor
    OSRAuthSecurityContext & operator=(OSRAuthSecurityContext&&) = delete;    // move assignment
};
```

Figure 6—An RAIL class for **SUBJECT_SECURITY_CONTEXT**

C++ Features... (Cont.)

(CONTINUED FROM PAGE 10)

```
//
// Start by capturing and locking the Subject Security Context... we're
// going to need it, regardless of the type of operation that we're
// processing.
//
OSRAuthSecurityContext securityContext(Request);

//
// Let's JUST check the primary token...
//
primaryToken = securityContext.PrimaryToken;

// more code here...
```

Figure 7—Instantiating the RAII Class

To take advantage of this RAII object, all we need to do is create an instance on the stack, like you see in *Figure 7*. Then you can use the class like you'd expect.

Now, for sure, there's nothing in this so far that you couldn't have accomplished with a helper function that takes (or returns) a pointer to some sort of "security context" structure and initializes it using the passed-in Request. The magic comes at function exit, shown in *Figure 8*.

```
Exit:
//
// Note the Security Context is unlocked and released automatically.
// See the definition for the class OSRAuthSecurityContext
//
if (INT_SUCCESS(status)) {
    WdfRequestComplete(Request, status);
}

return;
}
```

Figure 8—RAII does the cleanup

At the end of the function in which our **OSRAuthSecurityContext** is used, the tear-down is done automatically by the object's destructor. So, the security context is always properly unlocked and released. There's no chance of a leak. Our only problem is that we need to be sure we document that this is what's going on, for the potentially unsuspecting maintainers that come after us.

I don't know about you, but I call that both "wonderful" and "magic." Again, it's not something I need to do in every driver I write. But when I need it, it's nice to have the feature available.

#6 (Free Bonus!): Default Arguments

I promised you five features to try. But, since I started with one (strong type checking) that is pretty well-known and non-controversial, I'll give you one, last, extra-special bonus feature: default arguments.

```
void
InitializeMyThing(CHAR * StringToInit,
                 ULONG LengthToInit,
                 CHAR CharacterToUse = 0x00)
{
    memset(StringToInit, LengthToInit, CharacterToUse);
}
```

Figure 9—Example with an optional parameter

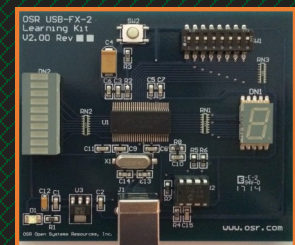
Default arguments can make coding certain functions very easy. Take the example shown in *Figure 9*. Again, please set aside your judgment as to whether this is a function you'd really want to put in your code... it's just an example.

(CONTINUED ON PAGE 12)

USB FX2 Learning Kit

Don't forget, the popular OSR USB FX2 Learning Kit is available in the OSR Store at:
<http://store.osr.com>.

The board design is based on the well-known Cypress Semiconductor USB FX2 chipset and is ideal for learning how to write Windows drivers in general (and USB specifically of course!). Even better, grab the sample WDF driver for this board, available in the Windows Driver Kit.



C++ Features... (Cont.)

(CONTINUED FROM PAGE 11)

In Figure 9, we have a function that takes three parameters. The last parameter, **CharacterToUse**, is optional. If it's not supplied when the function is called, a default value (0x00 in the example) will be used. And you can then invoke this function with, or without, that final optional argument as shown in Figure 10.

```
CHAR fred[FRED_LENGTH];
InitializeMyThing(fred, FRED_LENGTH);
InitializeMyThing(fred, FRED_LENGTH, 0xFF);
```

Most Modern C++ features (like the vast majority of the C++ Standard Library) are primarily applicable to user-mode code. But I hope you've seen here that there are a few Modern C++ features that can make your coding life easier, and perhaps even make your drivers more reliable.

Figure 10—Invoking a function with an optional parameter

It's worth noting that there are plenty of other C++ features that I like and that work in drivers that I haven't mentioned. For example **unique_ptr**, **std::tuple**, many of the functions in **std::algorithm**, and perhaps even some uses of lambda expressions. And I dream of someday being able to use **std::vector** and **std::map**.

I hope you'll try some of these C++ features that are new to you in driver projects of your own. If you do, head over to [NTDEV](#) and tell us about your experience.



Follow us!



OSR thanks C/C++ expert Giovanni Dicanio for taking the time to tech-review this article so that we didn't expose ourselves as C++ dilettantes. You can catch up with Gio via his blog at <https://blogs.msmvps.com/gdicanio/>. Gio is an author of several excellent C/C++ related courses that you can [find on Pluralsight](#). Don't forget that if you have an MSVC subscription, you probably also get free access to Pluralsight courses!

Why Engage
with OSR?

We Know What We Know And...We Know What we Don't Know

We are not experts in everything. We're not even experts in everything to do with Windows. But we think there are a few things that we do pretty darn well. We understand how the Windows OS works. We understand devices, drivers, and file systems on Windows. We're pretty proud of what we know about the Windows storage subsystem.

What makes OSR unique is that we can explain these things to your team, provide you new insight, and if you're undertaking a Windows system software project, help you understand the full range of your options. AND we also write kick-ass kernel-mode Windows code. Really. We do.

Why not fire-off an email and find out how OSR can help. If we can't help you, we'll tell you that, too.

Contact: sales@osr.com

New Options for Compile-Time Validation

Making VS Static Analysis Tools Work For You (Not Annoy You)

My C programming, and especially my attempts at using C++ “as a better C”, can usually use a lot of help. I admit it. My code quality can be inconsistent. Sometimes it’s because I’m not fully dedicated to a specific coding convention to use: Do I want to continue using **NULL** or should I switch to **nullptr**? Sometimes it’s because I’m initially focused on getting my device to work, and I take some messy shortcuts. That’s fine, except when those shortcuts get accidentally left in the code at Final Release. Stuff like this:

```
// Set the BME280 into NORMAL mode, with 16x oversampling for both
// temp and pressure.
writeBuffer[0] = BME280_REGISTER_CONTROL;
writeBuffer[1] = 0xB7;

status = Bme280WriteData(DevContext,
                        writeBuffer,
                        sizeof(writeBuffer));
```

OK, that’s not the worst code ever written. But WTF does *0xB7* mean?? Yes, the comment implies it means “16x oversampling for both temp and pressure” but shouldn’t we have used some sort of a symbolic constant instead of a literal hex value?

I’m also sort of famous for coding or cut/pasting a call to some function and then neglecting to check the return status. Yeah, I know: sloppy. I’m not proud of it. Sometimes my mind just starts working faster than my fingers.

Because of these things, I’ve learned to stop worrying and love static analysis tools.

What Tools Are Available?

Visual Studio 2019 incorporates multiple, highly configurable, static analysis tools that can be useful to driver developers. At OSR, we’ve found that the most success lies in tuning each of them properly and using them in combination.

The primary static analysis tool that Visual Studio provides is Microsoft Code Analysis, which everyone just refers to as Code Analysis or “CA.” This is the tool that started out as PREfast, was run via OACR, and eventually was integrated into the “new” C/C++ compiler. CA has a huge number of settings and options, including some that are specific to Windows drivers. In fact, the vast number of ever-changing settings is both one of the tool’s biggest advantages and biggest sources of frustration. We’ll talk more about these later.

A new static analysis option that’s become available starting with VS 2019 is [Clang-Tidy](#). Clang-Tidy is part of the LLVM tool suite. This tool, like Microsoft Code Analysis, examines your code for a wide variety of issues. Some of its checks have to do with common errors, others have to do with what it considers “best practices”, there are some checks that are purely stylistic in nature, and still others are focused on enforcing system-specific coding conventions (for example Clang-Tidy supports a set of checks for adherence to [Zircon kernel](#) coding conventions).

There is, of course, one more static analysis tool available to driver devs in the WDK: This is The Mother of All Static Analysis Tools, Static Driver Verifier (SDV). SDV aims at reviewing driver code for functional correctness, by simulating its interactions with the OS. We’re not going to discuss SDV in this article because (a) it provides a very different type of analysis from that provided by Code Analysis and Clang-Tidy, and (b) nobody at OSR has been able to get it to work – at all – in VS 2019. So, we’re going to ignore SDV for now ([See Sidebar: Real-Time Squiggles, Dots, Light Bulbs, Wrenches, Colors and Whatnot, P. 19](#)).

Selecting Code Analysis Rules and Options

You can tell VS to run Code Analysis automatically during your builds, by right-clicking on your project, and changing the settings under – not surprisingly – Code Analysis. You can see this in *Figure 1* (next page).

[\(CONTINUED ON PAGE 14\)](#)

VS Static Analysis Tools... (Cont.)

(CONTINUED FROM PAGE 13)

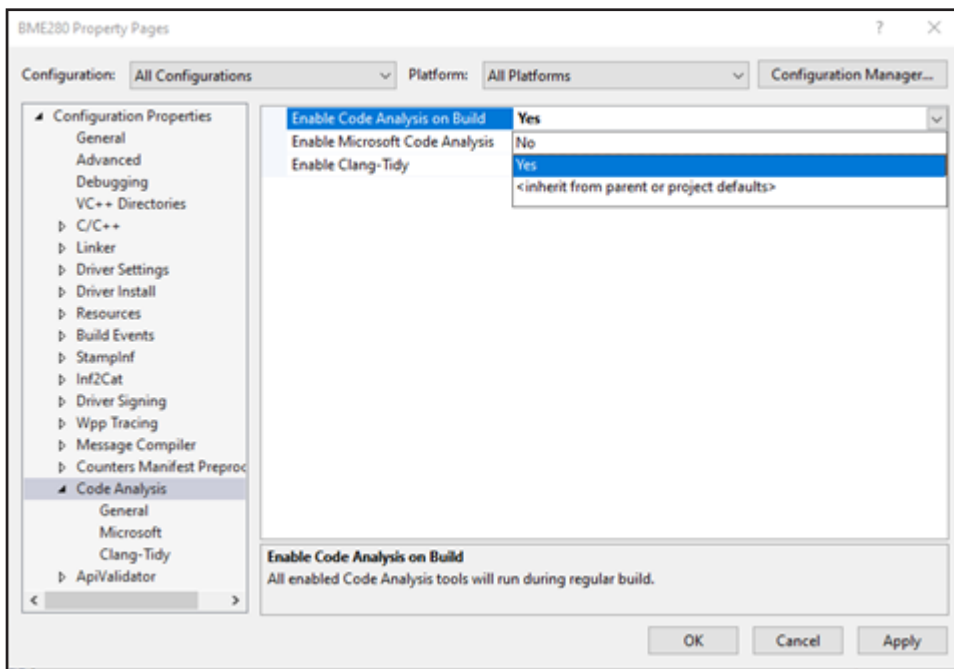


Figure 1—Enabling Code Analysis when building your driver

In this same page, you select whether you want VS to automatically run Microsoft Code Analysis, Clang-Tidy, or both. Of course, the same caveats apply to Code Analysis settings that apply to *any* project settings that you make in Visual Studio: Be aware of the Configuration and the Platform for which you're changing your settings.

Finally, note that in the Code Analysis settings section there are individual pages for indicating the specific configurations for Microsoft Code Analysis and Clang-Tidy. We'll talk about those in the sections describing each tool.

About Microsoft Code Analysis

One of the biggest frustrations that we encounter with Microsoft Code Analysis is that it changes, sometimes dramatically, with each release of Visual Studio. Sure, it's "always" been the case that as new versions of the compiler (or new versions of PREfast) are released, old spurious errors get fixed and new and different checks get implemented. But for the last several versions of VS (since the C/C++ compiler was rewritten, in fact) Microsoft Code Analysis has been decidedly temperamental. Sometimes it reports errors when compiling a given project. Sometimes it does not report errors when

(CONTINUED ON PAGE 15)

Now Available Online!
Live-Streamed via the Web

Need To Know WDF?

Tip: You can read all the articles ever published in *The NT Insider* and still not come close to what you will learn in one week in our [WDF](#) seminar. So why not join us?

Both Scott and Peter have in-depth knowledge and extensive hands-on experience writing device drivers. Their discussions about mistakes to avoid was as valuable as explaining Windows. This was the best training class that I have ever taken.

- Feedback from an attendee of THIS seminar

Seminar Outline and Information here: <http://www.osr.com/seminars/wdf-drivers/>

Next ONLINE presentation:
7-11 December 2020

VS Static Analysis Tools... (Cont.)

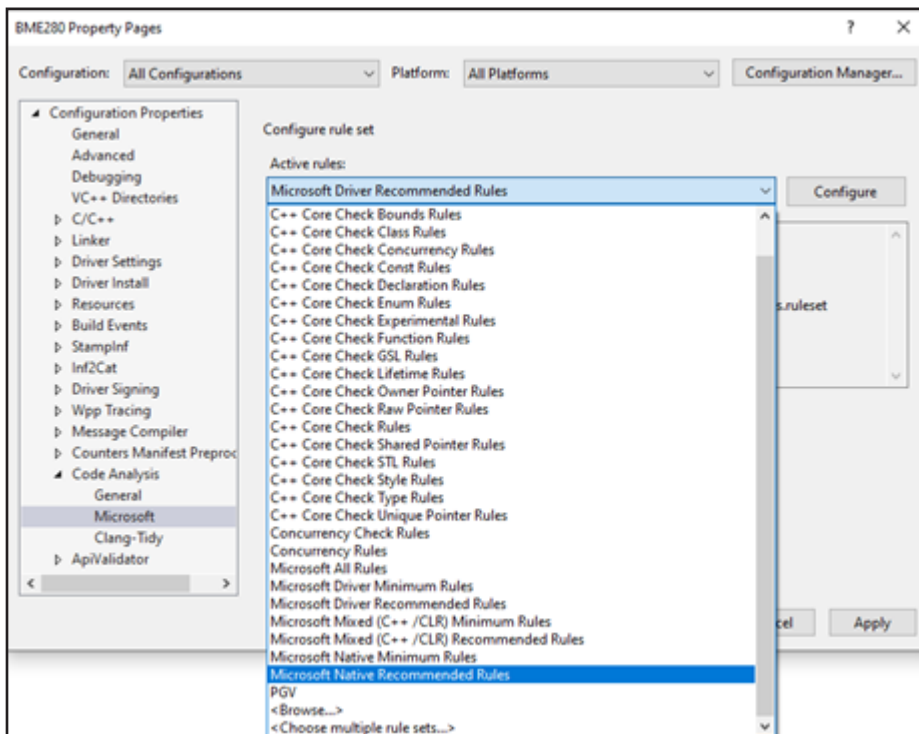
(CONTINUED FROM PAGE 14)

compiling that same project, with exactly the same configuration and platform. Sometimes it reports errors on one system, but not another. It's unpredictable. It's seemingly random. It's usually maddening.

On the other hand, with each release of VS, CA continues to add more, and sometimes even more useful, inspections. And false positives that existed in previous version of CA usually disappear. This alone is a good argument for keeping updated with the latest version of VS and the WDK.

The way you tell CA which inspections you want to be applied to your code is by selecting a set of rules. Microsoft provides a (large and unwieldy) set of rules collections, that provide various checks and levels of scrutiny.

For years we preached that you should make your drivers build "clean" (with /W4 and) selecting the CA rule set "Microsoft All Rules." We even had a slide that said this in our WDF seminar. Then, with some update of something or other, "All Rules" suddenly included checks for compliance with the [C++ Core Guidelines](#). While I'm certainly in favor of the C++ Core Guidelines as a general principle, I'm not a believer in every one of the rules that these constructs advocate. Not to mention the fact that many of the rules set forth by the Guidelines simply can't be applied to writing drivers for Windows ("I.10: Use [C++] exceptions to signal a failure to perform a required task" is the most obvious example).



So, how can you best use Microsoft Code Analysis and its vast array of rule sets in driver projects? Let me try to provide you some guidance that you may find helpful.

Selecting a Microsoft CA Rule Set to Start With

In the project property pages, under the Code Analysis section, in the Microsoft subsection, you'll find a drop-down box that contains the known rule sets that you can select. See Figure 2.

Note that you can select one or more rule sets here, and you can even create your very own rule set with your own, customized, sets of CA inspections to apply. If you do that... good luck. And remember to check it in someplace with your project.

Figure 2—Pick a rule set...any rule set

(CONTINUED ON PAGE 16)

Design & Code Reviews

You've Written Code—Did You Miss Anything?

Whether you're a new Windows driver dev or you've written dozens of drivers before, it's always hard to be sure you haven't missed something. Windows changes, WDF changes, security issues emerge. Best practices are a moving target.

Let OSR help! Our engineering team is 100% dedicated to Windows internals and driver development. Let us be the expert, second pair of eyes on your project... ensure it's done right!

Contact: Sales@osr.com

VS Static Analysis Tools... (Cont.)

[\(CONTINUED FROM PAGE 15\)](#)

Here at OSR, by default we set our driver projects to “Enable Code Analysis on Build” for every Configuration and Property that the project supports, and we select the “Microsoft Driver Recommended Rules” rule set. If you don’t see this rule set as an option, choose “Browse...” in the rule selection dialog and look in the “%ProgramFiles(x86)%\Windows Kits\10\CodeAnalysis” directory. You’ll find the driver-specific rule sets there.

It’s worth noting that the *driver-specific* inspections that are enabled by the “Microsoft Driver Recommended Rules” and “Microsoft Driver Minimum Rules” are identical. These rule sets only differ in terms of which C/C++ “native” rule sets are also applied. The “Driver Minimum” rule set uses the C/C++ native “Minimum” rules. The “Driver Recommended” rule set uses the C/C++ native “recommended” rules.

In general, Microsoft CA doesn’t usually spew a lot of false positives, and your driver should be able to pass the “Driver Recommended” rule set inspections without any warnings. But saying this is definitely not the same as saying that you should never suppress any warnings that Microsoft CA displays. You can see an example where CA gets it wrong in *Figure 3*.

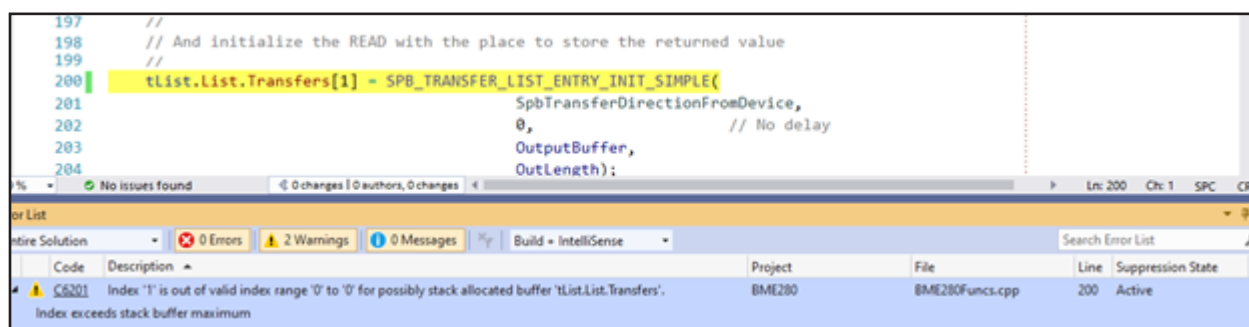


Figure 3—No. That warning is wrong

I was too lazy to include the code that creates the SPB_TRANSFER_LIST_AND_ENTRIES in the code clip, and unless you write SPB drivers you won’t care what it says in any case, but you can trust me what I tell you that it allocates two entries for the list and the CA warning is just flat-out wrong.

Fortunately, when CA gives you a spurious warning, you can right-click the error (directly in the VS Error List dialog) and tell VS to suppress the warning by putting a comment in the source code. When you suppress a CA warning, do yourself a favor and put a comment that includes the version of the WDK being used and the error type. The WDK version implies a version of VS, so you shouldn’t need to note both (*Figure 4*).

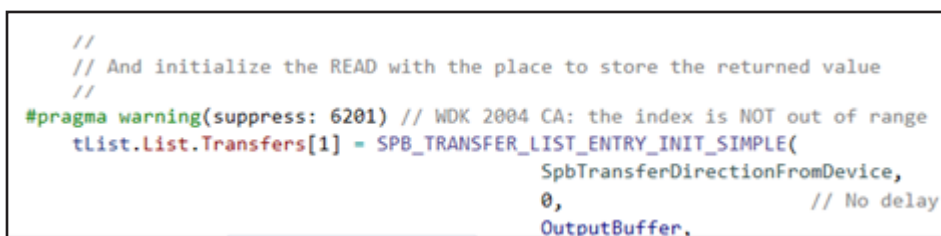


Figure 4—When you suppress a warning, note the WDK version

When we change the tool chain that we use for a given project (for example, if we update from VS 2017 to VS 2019 for a given release of a product) we always (try to) go back to the projects, remove all the warning suppressions, and run CA again on the project. We usually find that when a legitimately spurious error has been suppressed, CA will eventually get fixed and the suppression comment is no longer needed.

So, that’s what we do here at OSR by default on every build. And it’s what we recommend you do as well. But much there’s more you can do. Before releasing code for a project, we’ll usually do some builds with additional rule sets selected.

[\(CONTINUED ON PAGE 17\)](#)

VS Static Analysis Tools... (Cont.)

[\(CONTINUED FROM PAGE 16\)](#)

Going Further with Microsoft CA

Before we release our code to the world, we'll often do a few Microsoft CA builds using the "Microsoft All Rules" rule set. This can be useful for finding problems that won't otherwise be flagged, or for pointing out things (like marking parameters or variables **const**) that you might want to at least think about. Note that many of the inspections that Microsoft CA provides are duplicated by Clang-Tidy, and often in a more effective or useful way. We'll talk Clang-Tidy later in this article.

When we run "Microsoft All Rules" on our code, we usually find the Core Guidelines checks overwhelming unless we disable a few basic checks for things like C-style casts and uninitialized variables. We do this with **#pragma warning(disable : xxxx)** statements in the driver's master header file. You can see the suppressions that we start with for drivers in *Figure 5*.

```
#pragma warning(disable: 26438) // avoid "goto"
#pragma warning(disable: 26440) // Function can be declared "noexcept"
#pragma warning(disable: 26485) // No array to pointer decay
#pragma warning(disable: 26493) // C-style casts
#pragma warning(disable: 26494) // Variable is uninitialized
```

Figure 5—Suppressions we enable by default when running CA with "Microsoft All Rules"

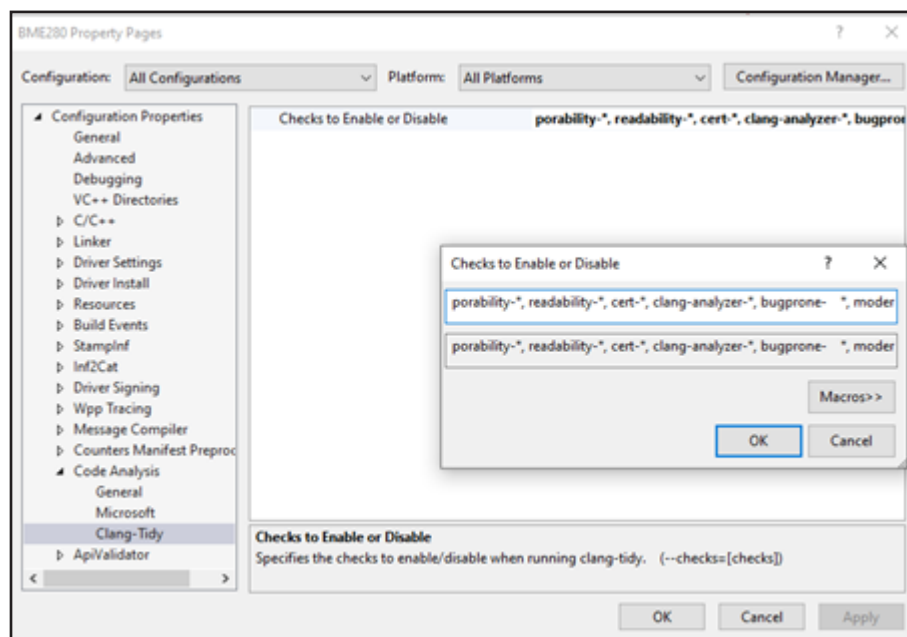
You may like or you may hate some of these suppressions (note, for example, that we embrace the use of **goto**). No matter, you'll almost certainly want to customize them to your chosen coding style, preferences, and practices.

In general, we do not bother trying to get our driver code to the point where "Microsoft All Rules" runs clean. Rather, we use the warnings that are displayed as suggestions. And once we've satisfied ourselves with any changes that we've made as a result of these suggestions, we switch the project back to running "Microsoft Driver Recommended Rules" on every build by default.

Even Better Code with Clang-Tidy

While running Microsoft CA with "All Rules" can be interesting, running Clang-Tidy can be both interesting and often extremely useful. Clang-Tidy is a very powerful, very comprehensive, and very smart static analyzer that is capable of inspections that enforce things ranging from coding style to correctness.

Unfortunately, VS doesn't provide any default Clang-Tidy inspections. Instead, you need to enter a list of the Clang-Tidy checks or groups of checks that you want to enable or disable. You do this in the Clang-Tidy subsection of the Code Analysis section of the project property dialog. See *Figure 6*.



[\(CONTINUED ON PAGE 18\)](#)

Figure 6—Enabling specific Clang-Tidy checks

VS Static Analysis Tools... (Cont.)

(CONTINUED FROM PAGE 17)

Fortunately, [the Clang-Tidy docs are pretty good](#) at explaining the various possible checks, and often provide examples of things the inspection checks for. Because these inspections can be so wide-ranging, your coding style will have a significant impact on the checks you want performed. To give you someplace to start, *Figure 7* has the list of checks that we usually enable by default at OSR.

porability-*, readability-*, cert-*, clang-analyzer-*, bugprone- *, modernize-*, misc-*,-modernize-use-trailing-return-type, -modernize-use-using, -modernize-avoid-c-arrays, -modernize-use-auto, -readability-implicit-bool-conversion, -readability-delete-null-pointer, -readability-simplify-boolean-expr, -readability-redundant-control-flow

Figure 7—The Clang-Tidy settings that we use at OSR

When you enable Clang-Tidy inspections, the outputs and warnings from those inspections appear in the Visual Studio Output window, not in the Error List window (as with Microsoft CA).

Like with Microsoft CA's "All Rules" rule set, when we run Clang-Tidy on (even) a well-written driver project we usually wind-up with a few dozen warnings. As a result, we almost never aim for zero warnings from Clang-Tidy. Rather, we run with Clang-Tidy analysis one or two times each release cycle, and take under advisement the suggestions it provides. I often find Clang-Tidy's observations particularly helpful when I'm modernizing an older or an inherited driver project, while I try to bring that driver's code up to a bit of a higher standard. Given that cross-platform code is becoming more important these days, one check that I love is that it flags non-portable path specifications for include files, where the case of a file name is different from the case that appears in the `#include` directive. You can see some examples of the type of warnings Clang-Tidy displays in *Figure 8*. (next page).

Go Boldly

Clang-Tidy and "Microsoft All Rules" giving you a bunch of warnings that you will not want to fix is *definitely not* a good reason for you to choose not to run these analyses. Consider running these checks like a mini in-office code review, but without the insults, rude remarks, and potential to impact your annual bonus. Who wouldn't want *that*?

(CONTINUED ON PAGE 19)

Now Available Online!
Live-Streamed via the Web

OSR's Corporate, Online Training

Save Money, Travel Hassles; Gain Customized Expert Instruction Remotely!

We can:

- Prepare and present a one-off, private, online seminar for your team to address a specific area of deficiency or to prepare them for an upcoming project.
- Design and deliver a series of offerings with a roadmap catered to a new group of recent hires or within an existing group.
- Work with your internal training organization/HR department to offer monthly or quarterly seminars to your division or engineering departments company-wide.

Contact: Seminars@osr.com

VS Static Analysis Tools... (Cont.)

(CONTINUED FROM PAGE 18)

```
F:\_Work\git_root\bme280\BME280\BME280Funcs.cpp(22,10): warning GA2B63E01: non-portable path to file ""BME280.h""; specified path differs in case from file name on disk [clang-diagnostic]
#include "Bme280.h"
      ^
      "BME280.h"
F:\_Work\git_root\bme280\BME280\BME280Funcs.cpp(68,30): warning G0B300A4C: 12 is a magic number; consider replacing it with a named constant [readability-magic-numbers]
    WCHAR    numString[12]; // big enough for any 32-bit number
                        ^
F:\_Work\git_root\bme280\BME280\BME280Funcs.cpp(113,40): warning G6731B54F: 10 is a magic number; consider replacing it with a named constant [readability-magic-numbers]
                        10,
                        ^
F:\_Work\git_root\bme280\BME280\BME280Funcs.cpp(367,31): warning G3A06AF62: 0xFF is a magic number; consider replacing it with a named constant [readability-magic-numbers]
    UCHAR    returnedChipId = 0xFF;
                        ^
F:\_Work\git_root\bme280\BME280\BME280Funcs.cpp(375,5): warning GC1A2668F: Value stored to 'status' is never read [clang-analyzer-deadcode.DeadStores]
    status = Bme280ReadRegisters(DevContext,
    ^
```

Figure 8—A random selection of Clang-Tidy warnings: Plenty of stuff to look at, and also probably to fix, in this driver

Static Analysis, FTW

It used to be that once you wrote and debugged your code, the only hope for improving it was either a thorough code review by your colleagues or your own code inspection undertaken when you return to a module after some time. Those activities still provide unmatched value. But static analysis tools, whether the ones that are built-in to Visual Studio or those provided by third party VS extensions, can go a long way toward improving code quality, readability, and maintainability.



Follow us!



Sidebar: Real-Time Squiggles, Dots, Light Bulbs, Wrenches, Colors, and Whatnot

It seems there are two very different types of devs: Those who revel in “interactive help as you type” and those that find the pop-ups, squiggles, and extreme chroma-coding distracting and annoying as hell. I am in the former category. I can use all the help I can get, and if I can get it sooner – like, as I’m typing – I figure that just makes my life easier.

Many years ago, when working on an enormous C# project, I learned the value of ReSharper. I discovered that, using it, I could write pretty respectable C# code. Since that time, I’ve tried to up my driver coding game by alternately using **ReSharper for C++** and **Visual Assist for C/C++**.

I used **Visual Assist** for my C/C++ projects happily for several years. I even [wrote an article](#) about why it was my VS extension of choice. But I recently switched to using **ReSharper for C++**. It’s purely personal preference, but in my opinion the ReSharper people have really upped their game in the last couple of versions when it comes to C++ code inspection and refactoring. No, the C++ support in **ReSharper** isn’t anywhere nearly as good as **ReSharper**’s support for C# (where ReSharper practically writes – or re-writes – your code for you). But it’s darn good.

Tools like **ReSharper** and **Visual Assist** can feel instantly overwhelming and frustrating to some users. In my experience, the key to happiness with using these tools lies in taking the time to configure and tune them the way you want them. This is also true for the VS IDE, of course. The problem can be that there are so many choices for formatting and code inspection, often with each choice having multiple options, that the tuning and configuring can be overwhelming (or, at least, tiresome). This is especially true if you’re forced to do this tuning and configuring by navigating a long list of check boxes and drop-down lists: Let’s see... do I want a space before *and* after member operators? Yes? No? Do I even know what this means? How many such options can I really consider in a row before I just quit and get coffee?

One thing that I really value about **ReSharper for C++** is that it allows you to tune and configure many options from the code editor, in real-time. This saves you the torture of navigating that long list of names and settings. The way this works is, when an issue is reported you right click it and **ReSharper** will allow you to tune its “severity” (and thus the type of visual annotation that it displays), or even to disable it (for a given occurrence with a comment, in the whole file with a comment, or even globally).

(SIDEBAR CONTINUED ON PAGE 20)

VS Static Analysis Tools... (Cont.)

(SIDEBAR CONTINUED FROM PAGE 19)

One thing I love most about **ReSharper for C++** is that, in addition to its own real-time inspections and formatting tricks, it has real-time support for Clang-Tidy static analysis inspections. Getting Clang-Tidy feedback at compile time (the way VS provides it) can be useful. But I love getting those hints in real-time in the IDE.

```
bool
EncryptCallback(void*    CallbackContext,
                uint64_t FinalSize,
                void*    StreamData,
                uint32_t StreamDataLength,
                bool      WriteEncrypted)
{
    SampSaOutputEngine* outputEngine = (SampSaOutputEngine*)CallbackContext;
    bool                (class) SampSaOutputEngine
    if (!outp
    //
```

So, for me, more squiggly lines and icons and funny chroma-coding the better. When the COVID Pandemic started and it became clear that I'd need to write actual code while at home and not merely spend countless hours surfing the internet, one of the first things I had to do was get **ReSharper** installed on my home system (I already had VS and the WDK) and get my config imported from my desktop system at work. If you've never used tools like **ReSharper** or **Visual Assist**, I urge you to give them a try.

For the avoidance of doubt: Since [quitting the Microsoft MVP program](#) last year, I pay the retail price for **ReSharper**. Neither I nor anyone at OSR has any relationship with, nor receive any compensation from, JetBrains (the company that makes **ReSharper**) or from Whole Tomato (the company that makes **Visual Assist**).

[\(RETURN TO ARTICLE\)](#)

Windows or Linux File Encryption? Let Us Help!

OSR's File Encryption Solution Framework (FESF)

We know that implementing per-file encryption solutions is challenging—and we've been doing it for more than 20 years!

Of course, it's also a challenge to convince devs and their managers of the difficulties they face. If you're new to Windows, you might be looking at the WDK examples and think "This looks easy!" If you're targeting Linux you might think "Hey, I can always start with one from an open source package."

Then you can spend months — MANY months — getting something to work. We run into this almost daily. Devs who are "one bug away" from getting their solution to work. But that "one bug" turns into "one more bug" and this goes on... for months.

Please don't let this be you. OSR can help save you a great deal of time and money and help make your Windows file encryption solution successful. OSR's latest toolkit—the [File Encryption Solution Framework \(FESF\)](#) builds on a time-tested infrastructure, but moves all the core development for a file encryption solution to USER mode. You don't need to be an expert in Windows file system or kernel programming, and your time is better spent defining "policy" of your solution (what you want to encrypt, when, and with what algorithm and key management) instead of wrestling with the subtle and painful nuances in filtering file systems.

Want to try a fully-functional evaluation of FESF for FREE? Just contact the OSR sales team and they'll get you started.

Contact: sales@osr.com

**Free FESF Eval
Available Now!**

Learning By Example

A Generic Device Class Filter Using WDF

How many times have you been working on a device or a driver project, seen some behavior in one of the existing Windows device stacks, and asked yourself “I wonder what’s going on with that particular sequence of I/O requests” or “What Control Codes, exactly, does that driver handle”?

Given that we don’t have the source code to most of the drivers that ship with Windows, getting these answers might seem daunting. But, fortunately, in Windows we have Filter Drivers.

In this article, we’re going to provide a brief overview of device Class Filters and present a tutorial walk-through of an example of a generic device Class Filter. As a part of this tutorial, we’ll describe the primary differences between WDF function drivers and WDF filter drivers, and how you receive, examine, and send Requests to the next device in the device stack.

The source code for the example driver we’ll be discussing is [available on GitHub](#).

Let’s start with some general background on Windows filter drivers.

Different Types of Filter Drivers

As you probably already know, a Filter Driver is a uniquely powerful Windows component that allows you to insert a “Filter Device Object” above or below a given Device Object in a Windows device stack. This allows you to provide a driver that observes, manages, or even modifies I/O operations as they are processed. If you’re having trouble remembering your Windows device stack concepts (PDOs, FDOs, and the like) [see the sidebar entitled Quick Review: PDOs and FDOs](#). It’s a quick read, and the review will probably do you good.

There are several different types of Filter Drivers, ranging from File System Filters to Filters for specific PnP device instances. In this article, we’re going to restrict our discussion to PnP device **Class** Filters. These are among the most common types of filter drivers written for Windows. They apply to entire installation class (i.e. category) of device(s). For example, a Class Filter could be used to filter all the CD-ROM devices in the system. Or all the Disk devices.

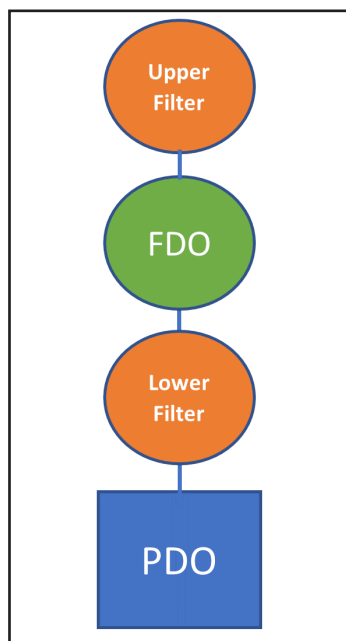


Figure 1—Filter Drivers (Upper and Lower) in a Device Stack

A Class Filter can be either be an *upper* filter or *lower* filter. Upper filters are instantiated *above the FDO* in the device stack, and therefore they see any I/O operations that are sent to a device before the function driver for that devices sees them. Lower filters are inserted below the FDO in the stack, and only get to handle I/O operations that are sent by the function driver “down” the stack towards the PDO. You can see what I’m talking about in *Figure 1*.

[\(CONTINUED ON PAGE 22\)](#)

Attend OSR Seminars Remotely!

What do our students say about their online seminar experience with OSR?

— I would *absolutely recommend* the online seminar experience to others. [Instructor] did an excellent job of making sure we (remote attendees) felt like we were part of the group. I greatly appreciate the opportunity to attend the seminar online, as it made logistics of attending much simpler.

GenFilter... (Cont.)

(CONTINUED FROM PAGE 21)

Class Filters are installed the way any driver is, using an INF file that copies the driver onto a local disk (typically to the `%windir%\system32\drivers` directory) and makes a series of entries in the Registry. The Device Install Class to be filtered is identified by a Device Class GUID. You can find [a list of interesting and useful Device Class GUIDs online](#). Each Device Install Class that's known to Windows (regardless of whether any such devices are currently attached to the system) appears in the Registry under `HKLM\System\CCS\Control\Class`, as shown in *Figure 2*. To indicate that you have a driver that wants to filter a given class of device, all you need to do (aside from installing your driver in the usual way) is make an "UpperFilters" or "LowerFilters" entry in the Registry under the Device Install Class that you want to filter. This will cause the PnP Manager to instantiate your filter either above (for an upper filter) or below (for a lower filter) the function driver's FDO for that class. But... WAIT! We're getting a bit ahead of ourselves. We'll look at the INF file to install our filter later in this article.

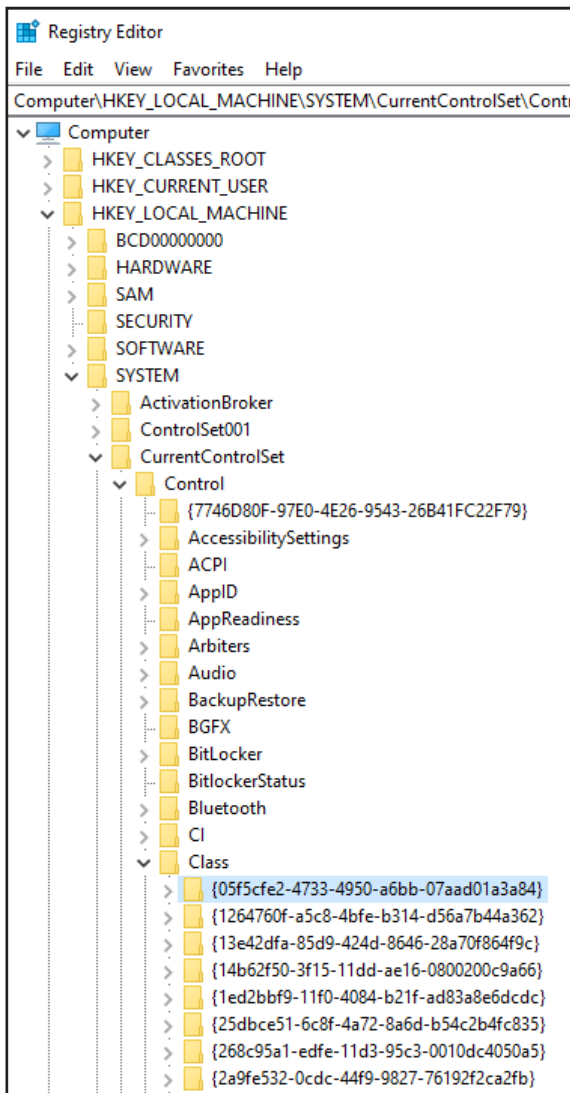


Figure 2—The start of the list of Device Install Classes in the Registry

These default behaviors make writing WDF filter drivers particularly easy and pleasant. But perhaps *the* most notable behavior change is the last one listed above: The Framework automatically sends any I/O Requests that your filter driver does not handle down the stack. What this means for you, as a driver dev writing a filter driver, is that you only need to write code to handle

WDF Filters: A Brief Overview

Two of the primary advantages that WDF provides over older Windows driver models are the overall simplicity/organization of the model and the fact WDF does much of the common (and sometimes complex) grunt work that writing a driver on Windows requires. This includes, you will recall, providing an integrated state machine for PnP and Power Management handling. These advantages are perhaps even more apparent and welcome when writing a filter driver using WDF than when writing a function driver.

A WDF filter driver is structured identically to a function driver. In **DriverEntry**, the driver creates its **WDFDRIVER** Object and connects with the Framework. For a filter driver, the **EvtDriverDeviceAdd** Event Processing Callback is called when the PnP Manager is in the process of enumerating a device that the driver has been specified to filter.

Within **EvtDriverDeviceAdd**, the filter driver informs the Framework that it is a filter (as opposed to a function driver) by calling the function **WdfFdoInitSetFilter**. Calling **WdfFdoInitSetFilter** causes the Framework to change many of its default function driver specific behaviors to filter driver specific behaviors. These changes include:

- When a filter driver creates its **WDFDEVICE** Object, the Framework will copy the I/O Type (such as Buffered I/O or Direct I/O) and Device Type from the Device Object that's being filtered to the filter driver's newly created **WDFDEVICE** Object.
- The Framework will (by default) set a filter driver to NOT be the Power Policy Owner for the stack.
- **WDFQUEUES** that are created by a filter driver will default to being non-power managed.
- The Framework will (again, by default) automatically send any Create, Cleanup, and Close Requests received by a filter driver to the underlying device stack.
- The Framework will automatically send down the device stack any Request types that a filter driver does not specifically handle.

(CONTINUED ON PAGE 23)

GenFilter... (Cont.)

[\(CONTINUED FROM PAGE 22\)](#)

Request types that your filter is interested in. If you only care about Read Requests, then you only need to write code that handles reads. If any other type of Request is sent to your filter driver, such as a Write or a Device Control, the Framework will automatically send that Request down the device stack to your Local (In-Stack) I/O Target. This relieves you from having to write lots of useless plumbing and allows you to focus on the work that you want to do.

In addition to specifying your device as a filter device in **EvtDriverDeviceAdd**, you will also need to create your WDFDEVICE Object in the usual way. And before leaving **EvtDriverDeviceAdd** you will also create any WDFQUEUES you want to use for receiving Requests, bearing in mind that you only need to provide Queues and EvtIo Event Processing Callbacks for those Requests that your driver wants to handle.

The only other Event Processing Callbacks that a WDF filter driver typically implements are the Callbacks related to Request processing. These are typically the EvtIo Event Processing Callback that process Requests types of interest that are presented from the Queue(s) that you created.

Let's now turn our attention to looking at the code for our generic filter driver sample. *Note that, in the interest of saving space, we've removed many of the comments and DbgPrint statements from the code shown in this article.* So, if you want to play with this code, we urge you to get the version from [GitHub](#) (and not just cut/paste the code from this article).

GenFilter: The Generic WDF Filter Driver

We wrote the Generic WDF Filter Driver to demonstrate the wonderful simplicity and usefulness of WDF filter drivers. It is a fully working example of a WDF class filter that you can build and use immediately, perhaps with a few customizations, to monitor or observe the I/O operations that are processed in a given device stack. By default, the INF file for this filter is set to filter CDROM Class devices. To filter another class of device, all you need to do is change the Class Name and GUID in the INF file. We'll talk more about the INF file later in this article.

[\(CONTINUED ON PAGE 24\)](#)

Now Available Online!
Live-Streamed via the Web

Need To Know WDF?

Tip: You can read all the articles ever published in *The NT Insider* and still not come close to what you will learn in one week in our [WDF](#) seminar. So why not join us?

Both Scott and Peter have in-depth knowledge and extensive hands-on experience writing device drivers. Their discussions about mistakes to avoid was as valuable as explaining Windows. This was the best training class that I have ever taken.

- Feedback from an attendee of THIS seminar

Seminar Outline and Information here: <http://www.osr.com/seminars/wdf-drivers/>

Next ONLINE presentation:
7-11 December 2020

GenFilter... (Cont.)

[\(CONTINUED FROM PAGE 23\)](#)

GenFilter: DriverEntry

Like most WDF drivers, GenFilter doesn't do much in its **DriverEntry** routine. Its activities are restricted to instantiating a WDFDRIVER Object and thus getting connected with the Framework. You can see the code in *Figure 3*.

```
extern "C"
NTSTATUS
DriverEntry(PDRIVER_OBJECT DriverObject,
            PUNICODE_STRING RegistryPath)
{
    WDF_DRIVER_CONFIG config;
    NTSTATUS status;

    #if DBG
        DbgPrint("GenFilter...Compiled %s %s\n",
                __DATE__,
                __TIME__);
    #endif

    //
    // Initialize our driver config structure, specifying our
    // EvtDeviceAdd event processing callback.
    //
    WDF_DRIVER_CONFIG_INIT(&config,
                          GenFilterEvtDeviceAdd);

    //
    // Create our WDFDEVICE Object and hook-up with the Framework
    //
    status = WdfDriverCreate(DriverObject,
                            RegistryPath,
                            WDF_NO_OBJECT_ATTRIBUTES,
                            &config,
                            WDF_NO_HANDLE);

    if (!NT_SUCCESS(status)) {
        goto done;
    }

    status = STATUS_SUCCESS;

done:
    return status;
}
```

Figure 3—DriverEntry

There's really not much to say about the code in **DriverEntry**. You can see the GenFilter doesn't specify any WDF_OBJECT_ATTRIBUTES, because it can't change the default parent of the **WDFDRIVER** Object that it's creating, it doesn't need to provide Cleanup or Destroy Event Processing Callbacks, and it doesn't need to specify a context for our WDFDRIVER Object. As you can see, GenFilter uses **DbgPrint** messages to display its output in the debugger. If the filter driver were using WPP Tracing, it would need to call WPP_INIT_TRACING here in **DriverEntry** and also provide a **Destroy** callback for our **WDFDRIVER** Object in which it called WPP_CLEANUP.

[\(CONTINUED ON PAGE 25\)](#)



THE NT INSIDER - Hey...Get Your Own!

Just visit The NT Insider page and subscribe with your email — and you'll get notification whenever we release a new issue of The NT Insider.

GenFilter... (Cont.)

(CONTINUED FROM PAGE 24)

GenFilter: EvtDriverDeviceAdd

Again, the processing in this callback is about as simple as it could possibly be. GenFilter starts by calling **WdfFdoInitSetFilter**, to inform the Framework that it wants to instantiate a **WDFDEVICE** Object for a filter device and not a functional device. Don't let the name of this function confuse you. Even though it's called **WdfFdoInit** it takes as input a pointer to the **WDFDEVICE_INIT** structure that's passed in the **DeviceInit** parameter of your **EvtDriverDeviceAdd** Callback. By using the name **FdoInit** instead of **DeviceInit** the Framework is trying to emphasize that this operation is only valid for filter and/or function device objects (that is, this function is not valid for PDOs). If you think that this is an annoying and potentially confusing quirk of Framework naming, I won't disagree with you. But nobody asked our opinion, so let's move on.

Next, GenFilter specifies a context area (**GENFILTER_DEVICE_CONTEXT**) for the Framework to associate with the **WDFDEVICE** instance. And then the driver calls **WdfDeviceCreate** to instantiate the **WDFDEVICE** Object.

Finally, before leaving **EvtDriverDeviceAdd**, the driver specifies EvtIo Event Processing Callbacks for the Request types that it's interested in handling, and creates a single, default, **WDFQUEUE**. In the GenFilter example, we specified EvtIo callbacks for Read, Write, and Device Control operations – just to show how it would be done. However, remember what we said earlier: You only need to specify EvtIo callbacks for those Request types that your filter wants to examine. Any Request types that you don't provide an EvtIo Event Processing Callback for will be automatically sent by the Framework to the next Device Object in the device stack. You can see the code for GenFilter's **EvtDriverDeviceAdd** in *Figure 4*.

```
NTSTATUS
GenFilterEvtDeviceAdd(WDFDRIVER Driver,
                     PWDFDEVICE_INIT DeviceInit)
{
    NTSTATUS status;
    WDF_OBJECT_ATTRIBUTES wdfObjectAttr;
    WDFDEVICE wdfDevice;
    PGENFILTER_DEVICE_CONTEXT devContext;

    //
    // Indicate that we're creating a FILTER Device, as opposed to a FUNCTION Device.
    //
    WdfFdoInitSetFilter(DeviceInit);

    WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE(&wdfObjectAttr, GENFILTER_DEVICE_CONTEXT);

    status = WdfDeviceCreate(&DeviceInit, &wdfObjectAttr, &wdfDevice);

    if (!NT_SUCCESS(status)) {
        goto done;
    }

    devContext = GenFilterGetDeviceContext(wdfDevice);
    devContext->wdfDevice = wdfDevice;

    //
    // Create our default Queue -- This is how we receive Requests.
    //
    WDF_IO_QUEUE_CONFIG_INIT_DEFAULT_QUEUE(&ioQueueConfig, WdfIoQueueDispatchParallel);

    //
    // Specify callbacks for those Requests that we want this driver to "see."
    //
    ioQueueConfig.EvtIoRead = GenFilterEvtRead;
    ioQueueConfig.EvtIoWrite = GenFilterEvtWrite;
    ioQueueConfig.EvtIoDeviceControl = GenFilterEvtDeviceControl;

    status = WdfIoQueueCreate(devContext->wdfDevice, &ioQueueConfig, WDF_NO_OBJECT_ATTRIBUTES, WDF_NO_HANDLE);

    if (!NT_SUCCESS(status)) {
        goto done;
    }

    status = STATUS_SUCCESS;

done:
    return status;
}
```

Figure 4—GenFilter EvtDriverDeviceAdd

(CONTINUED ON PAGE 26)

GenFilter... (Cont.)

[\(CONTINUED FROM PAGE 25\)](#)

GenFilter: EvtIoDeviceControl

Next, let's have a look at how GenFilter handles device control (that is, IOCTL) Requests that it receives.

To demonstrate how we might search for a given operation, the GenFilter example examines each device control Request that it receives. If the Control Code of the received device control is **IOCTL_YOU_ARE_INTERESTED_IN** (an IOCTL that we made up and defined in the driver's header file for this example) then the driver:

- Prints a message in the debugger
- Calls the function **GenFilterSendWithCallback**. This function sets an **EvtWdfRequestCompletionRoutine** Event Processing Callback that will be invoked when the Request is complete, and sends the Request to the underlying device stack, using **WdfRequestSend**, specifying asynchronous processing.
- When the **EvtWdfRequestCompletionRoutine** Event Processing Callback runs, prints out the status of the Request and completes it.

Of course, in a real filter driver, you could do anything you wanted with the Request(s) that you're interested in. This includes examining, or even changing, the data associated with the Request (before or after the Request has been processed) or sending the Request a different, or even an additional, I/O Target. It's up to you.

For those device control Requests that GenFilter receives that have a control code that we are *not* interested in (that is, in our example, the control code is not **IOCTL_YOU_ARE_INTERESTED_IN**) the driver calls **GenFilterSendAndForget** to send the Request to the underlying device stack and do no further processing on the Request.

[\(CONTINUED ON PAGE 27\)](#)

Windows Internals & Software Drivers

For SW Engineers, Security Researchers & Threat Analysts

Read what your colleagues have to say about this seminar. We can't put it any better!

Scott showed a strong mastery of Windows internals and an incredible knowledge base that he was capable of passing on to us.

Scott's knowledge in everything was incredible. Not only was he able to answer every question students had, but we also knew the complete history of why things were designed a certain way and how they worked!

Mr. Noone was definitely the best instructor I have had in my 20 years of being a programmer. He has a real knack for explaining things and added immense value to the slides and demos

"The instructor is extremely knowledgeable regarding Windows internals. He has the communications skills to provide an informative, in-depth seminar with just the right amount of entertainment value."

- Feedback from attendees of THIS seminar

Next Presentation:

ONLINE 25-29 January 2021

GenFilter... (Cont.)

[\(CONTINUED FROM PAGE 26\)](#)

You can see the code for **EvtIoDeviceControl** Event Processing Callback in *Figure 5*, and the code for **GenFilterSendWithCallback** and **EvtWdfRequestCompletionRoutine** in *Figure 6*. We'll show you the code for **GenFilterSendAndForget** in the next section.

```

VOID
GenFilterEvtDeviceControl(WDFQUEUE Queue,
                        WDFREQUEST Request,
                        size_t OutputBufferLength,
                        size_t InputBufferLength,
                        ULONG IoControlCode)
{
    PGENFILTER_DEVICE_CONTEXT devContext;

    devContext = GenFilterGetDeviceContext(WdfIoQueueGetDevice(Queue));

    UNREFERENCED_PARAMETER(OutputBufferLength);
    UNREFERENCED_PARAMETER(InputBufferLength);

    //
    // We're searching for one specific IOCTL function code that we're interested in
    //
    if (IoControlCode == IOCTL_YOU_ARE_INTERESTED_IN) {
#ifdef DBG
        DbgPrint("GenFilterEvtDeviceControl -
- The IOCTL we're looking for was found! Request 0x%p\n",
Request);
#endif
        //
        // Do something useful.
        //

        //
        // We want to see the results for this particular Request... so send it
        // and request a callback for when the Request has been completed.
        GenFilterSendWithCallback(Request, devContext);

        return;
    }

    GenFilterSendAndForget(Request,
                          devContext);
}

```

Figure 5—EvtIoDeviceControl

```

---

VOID
GenFilterSendWithCallback(WDFREQUEST Request, PGENFILTER_DEVICE_CONTEXT DevContext)
{
    NTSTATUS status;

    WdfRequestFormatRequestUsingCurrentType(Request);

    WdfRequestSetCompletionRoutine(Request, GenFilterCompletionCallback, WDF_NO_SEND_OPTIONS) {
        status = WdfRequestGetStatus(Request);
        WdfRequestComplete(Request, status);
    }

    //
    // when we return the Request is always "gone"
    //
}

VOID
GenFilterCompletionCallback(WDFREQUEST Request,
                          WDFIOTARGET Target,
                          PWDF_REQUEST_COMPLETION_PARAMS Params,
                          WDFCONTEXT Context)
{
    NTSTATUS status;
}

```

[\(CONTINUED ON PAGE 28\)](#)

GenFilter... (Cont.)

(CONTINUED FROM PAGE 27)

```

auto*    devContext = (PGENFILTER_DEVICE_CONTEXT)Context;

UNREFERENCED_PARAMETER(Target);
UNREFERENCED_PARAMETER(devContext);

DbgPrint("GenFilterCompletionCallback: Request=%p, Status=0x%x; Information=0x%x\n",
        Request,
        Params->IoStatus.Status,
        Params->IoStatus.Information);

status = Params->IoStatus.Status;

//
// Potentially do something interesting here
//
WdfRequestComplete(Request, status);
}

```

Figure 6—Send with Completion Callback

Perhaps the most interesting (and important) thing to note in *Figure 6* is in the **GenFilterSendWithCallback** function. Note that in this function, **GenFilter** checks the return value from **WdfRequestSend** and if it is **FALSE**, it calls **WdfCompleteRequest** for the Request. **WdfRequestSend** returns **FALSE** when it is unable to deliver the provided Request to the indicated I/O Target. So, **WdfRequestSend** returning **false** means that the send operation itself failed, the Request never reached the target device/driver. As a result, when **FALSE** is returned, the driver that called **WdfRequestSend** still “owns” the Request (and is therefore responsible for completing it).

Because **GenFilter** provides an **EvtWdfRequestCompletionRoutine** Callback, when **WdfRequestSend** succeeds, the filter driver will eventually be called back at its **EvtWdfRequestCompletionRoutine** Callback. There it re-gains ownership of the Request after the drivers in the underlying device stack have completed it. As mentioned earlier, in our example all we do is print a message in the debugger and complete the Request.

GenFilter: EvtIoRead and EvtIoWrite

We’ve included processing for **EvtIoRead** and **EvtIoWrite** just as examples of how to send a Request along with no further processing. Each of these routines simply prints out a message in the debugger and calls **GenFilterSendAndForget**. The code for **EvtIoRead** is in *Figure 7*.

(CONTINUED ON PAGE 29)

OSR's Debugging & Problem Analysis Service

We'll Root-Cause Crashes or Hangs so You Don't Have To!

Give me your tired, your poor, your huddled masses yearning to breathe free...OMG...that's exactly what [OSR's Debugging & Problem Analysis Service](#) is all about! Perhaps you're frustrated with a crash of your own kernel-mode software. Or you just dropped a beta on your customers and their CEO's system now hangs with your product installed. Or you manage several thousand PCs and have a handful of unrelated crashes that rear their ugly heads on a couple of subsets of systems in your domain....and you don't know who or what to blame. OSR can help.

We'll take your crash dumps and for a fixed-price, analyze and attempt to root-cause the issue, and report our findings. It's one of our most popular and valuable services. Don't bang your head against your keyboard any longer...it's just not healthy (or useful)!

Contact: sales@osr.com

GenFilter... (Cont.)

[\(CONTINUED FROM PAGE 28\)](#)

```
VOID
GenFilterEvtRead(WDFQUEUE Queue,WDFREQUEST Request,size_t Length)
{
    PGENFILTER_DEVICE_CONTEXT devContext;

    UNREFERENCED_PARAMETER(Length);

    #if DBG
        DbgPrint("GenFilterEvtRead -- Request 0x%p\n",Request);
    #endif

    GenFilterSendAndForget(Request,devContext);
}
```

Figure 7—GenFilter’s EvtIoRead Event Processing Callback

And the code for **GenFilterSendAndForget** appears in *Figure 8*.

```
VOID
GenFilterSendAndForget(WDFREQUEST Request,PGENFILTER_DEVICE_CONTEXT DevContext)
{
    NTSTATUS status;

    WDF_REQUEST_SEND_OPTIONS sendOpts;

    WDF_REQUEST_SEND_OPTIONS_INIT(&sendOpts,WDF_REQUEST_SEND_OPTION_SEND_AND_FORGET);

    if (!WdfRequestSend(Request,WdfDeviceGetIoTarget(DevContext->WdfDevice),&sendOpts)) {
        status = WdfRequestGetStatus(Request);
        #if DBG
            DbgPrint("WdfRequestSend 0x%p failed - 0x%x\n",Request,status);
        #endif
        WdfRequestComplete(Request,status);
    }
}
```

Figure 8—SendAndForget

Setting **WDF_REQUEST_SEND_OPTION_SEND_AND_FORGET** tells the Framework that when the driver sends this Request to the indicated I/O Target, the sending driver doesn’t want to see the Request again. There is no completion callback. When a driver sends a Request with **_SEND_AND_FORGET** it is the logical equivalent of that driver calling **WdfRequestComplete** for that driver. But, once again, note that the driver checks the return value from **WdfRequestSend**. As we saw previously, if the return value is **FALSE**, the send operation itself failed to deliver the Request to the indicated I/O Target. Therefore, the driver still owns the Request and is responsible for completing it.

Pretty Simple, Right?

So, that’s our generic filter sample driver. If you download the project from GitHub, you’ll see that there’s really no more to it than we’ve described above.

One Last Item: Installation

Back in the old days, it used to be the case that installing WDF Filters could be very tricky. This is because we needed to bring the co-installer along with the driver that we were installing. However, in recent versions of WDF (specifically, since Windows 8.1) we no longer need to provide the co-installer with WDF drivers. This is because (a) WDF drivers are “always” able to use a newer version of the Framework than that with which they were built, and (b) each OS version ships with (and receives updates to) the most up-to-date version of the Framework that will work on that version of the OS.

This makes installing WDF class filters extremely simple. In fact, you can create a trivial “right click” INF file to install your filter, as shown in *Figure 9*.

```
[Version]
Signature = "$Windows NT$"
Class = %ClassNameToFilter% ; Be sure the class NAME and ...
```

[\(CONTINUED ON PAGE 30\)](#)

GenFilter... (Cont.)

(CONTINUED FROM PAGE 29)

```

ClassGUID    = %ClassGUIDToFilter%    ; ... the class GUID agree.
Provider     = %Provider%
DriverVer    =
CatalogFile  = GenFilter.cat

[DefaultInstall.NT]
CopyFiles    = @GenFilter.sys
AddReg       = GenFilter.AddReg

[DestinationDirs]
DefaultDestDir = 12

[GenFilter.AddReg]
HKLM, System\CurrentControlSet\Control\Class\%ClassGUIDToFilter%, UpperFilters, 0x00010008, %
DriverName%

[DefaultInstall.NT.Services]
AddService = GenFilter, , GenFilter.Service.Install

[GenFilter.Service.Install]
DisplayName = %ServiceName%
Description = %ServiceDescription%
ServiceBinary = %12%\%DriverName%.sys           ;%windir%\system32\drivers\
ServiceType  = 1                               ;SERVICE_KERNEL_DRIVER
StartType    = 0                               ;SERVICE_BOOT_START
ErrorControl = 1                               ;SERVICE_ERROR_NORMAL

[SourceDisksFiles]
GenFilter.sys=1

[Strings]
ClassGUIDToFilter    = "{4d36e965-e325-11ce-bfc1-08002be10318}"
ClassNameToFilter    = "CDROM" ; MUST AGREE WITH ABOVE
Provider             = "OSR Open Systems Resources, Inc."
ServiceDescription    = "Generic Upper Filter"
ServiceName          = "GenFilter"
DriverName           = "GenFilter"
DiskId1              = "Genric Upper Filter Installation Disk"

```

Figure 9—"Right Click" INF file for Generic Filter

In Figure 9, you can see that we've setup our INF file so that our Generic Filter will be installed as an upper filter of all CDROM class devices. These are both definitively indicated in the **GenFilter.AddReg** section, where we make the necessary Registry entry under the Class GUID for the CDROM devices. You can see the resulting entry in *Figure 10*.

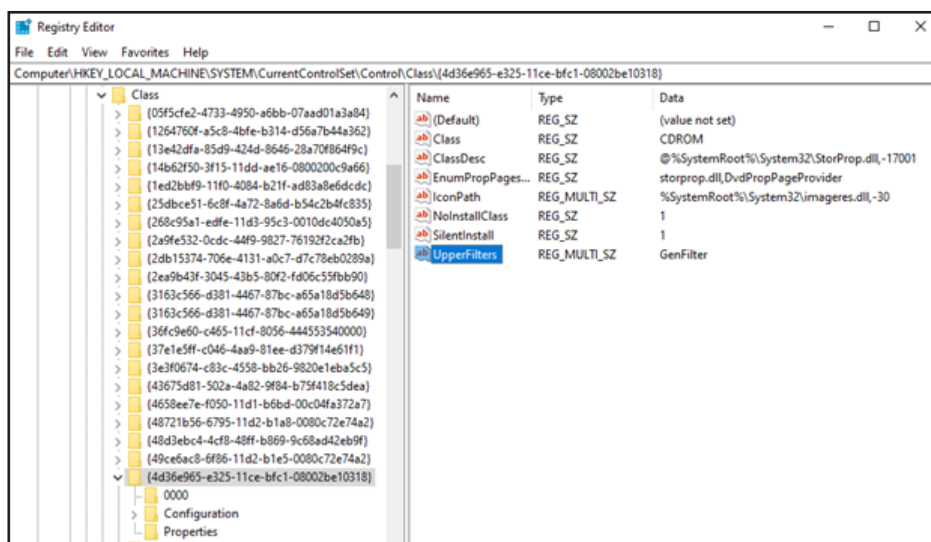


Figure 10—GenFilter Setup as an Upper Filter in Registry



Follow us!



Pool Allocator... (Cont.)

Sidebar: Quick Review: PDOs and FDOs

Let's start at the beginning, with PDOs and FDOs, because this is actually where most of the trouble starts.

As a rule, every device in Windows is discovered through the Plug-and-Play (PnP) process. The only exceptions to this rule are (a) software-only drivers that we refer to as “kernel services”, and (b) super-ancient hardware drivers that use the original Windows NT model. “Kernel services” are sometimes also referred to as “legacy style software drivers” or “NT V4 style software drivers.” The drivers in these exception categories create their Device Objects within their DriverEntry entry point. Lots of folks (including us) write kernel services to do things like monitor process creation, watch for registry changes, or provide other sorts of services from kernel mode. For the purposes of this article, we're going to ignore all drivers that aren't started by PnP.

So... as we said... as a rule, every device in Windows is discovered through the PnP process. This is true regardless of whether the device lives on a dynamically enumerable bus (such as PCIe or USB) or on a bus where the attached devices can't be discovered at run time (such as I2C or SPI). When a bus driver enumerates a device on its bus, it creates a Device Object that represents the physical instance of the discovered device on its bus. In WDF, the bus driver creates this Device Object using the function WdfDeviceCreate. This Device Object, created by the bus driver, is referred to as a Physical Device Object, or PDO for short.

As part of the overall PnP process, the PnP Manager queries the bus driver for a list of its “child devices.” If the bus driver has discovered any devices on its bus, it replies with a list of pointers to the PDOs that it has previously created to represent those devices.

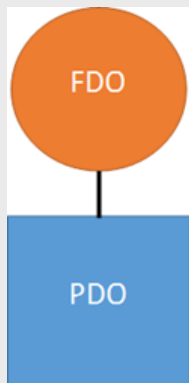


Figure A—Basic Device Stack

For each PDO returned from a bus driver (with some limited, special-case, exceptions), Windows attempts to find and start a driver that will be responsible for the functional aspects of the device. This is the function driver. It's at this point that WDF function drivers are called at their EvtDriverDeviceAdd Event Processing Callback. Within this Callback, a WDF function driver creates the Device Object (using WdfDeviceCreate) that represents the functional aspect of the device. This Device Object, created by the function driver, is referred to as the Functional Device Object or FDO for short.

After the FDO has been created, it is attached to the underlying PDO that was previously created by the bus driver. The result is a pair of Device Objects that together represent (a) the physical presence of a device on a given bus (PDO), and (b) the functional aspect of that device (the FDO). This pair forms the basis of the Device Stack and is shown in Figure A. For the sake of simplicity, we're ignoring filter drivers and their Device Objects in this discussion.

It's important to note that, regardless of how a device is accessed, any I/O operations that are sent to a device will always enter at the top of the Device Stack in which the device appears. In other words, I/O requests will always go to the FDO first. This makes sense, because it's the function driver (the one that created the FDO) that is responsible for the functional aspects of the device. And it's most typically only the function driver (and not the bus driver) that knows how to process I/O requests for the device. In fact, the bus driver is rarely involved in processing typical I/O operations.

[\(RETURN TO ARTICLE\)](#)

OSR Seminar Schedule

Seminar	Dates	Location
WDF Drivers I: Core Concepts	7-11 December 2020	ONLINE
Internals & Software Drivers	25-29 January 2021	ONLINE
Developing File Systems Mini-Filters	8-12 March 2021	ONLINE
Kernel Debugging & Crash Analysis	TBD	ONLINE

More Dates/Locations Available—See [website](#) for details

[Join OSR's Seminar Update Mailing List](#)

OSR Seminars

We Practice What We Teach For a Reason

When we say “we practice what we teach”, this mantra directly translates into the value we bring to our seminars. But don’t take our word for it...

—Like drinking from a fire hose of enlightenment.

—[instructor] was amazing and didn't hesitate to answer questions in the classroom to clarify concepts and functionality that may have been complex to understand. It was more than a pleasure to have his perspective and anecdotes to provide context to the material.

—[Instructor] was incredibly on the ball—on the evening of the first day, I had a considerable list of questions which I emailed him. I received a response about an hour after, and I didn't even expect a reply until class next day.

—The seminar gave me what I needed to hit the ground running on converting WDM drivers to KMDF.

—I would absolutely recommend the online seminar experience to others. [Instructor] did an excellent job of making sure we (remote attendees) felt like we were part of the group. I greatly appreciate the opportunity to attend the seminar online, as it made logistics of attending much simpler.

-Recent attendees of OSR seminars



[THE NT INSIDER](#) - Hey...Get Your Own!

Just visit The NT Insider page and subscribe with your email — and you'll get notification whenever we release a new issue of The NT Insider.

Private Training ONLINE

A private, online seminar format allows you to:

- **Get project-specific questions answered.** OSR instructors have the expertise to help your group solve your toughest roadblocks.

- **Customize your seminar.** We know Windows drivers and file systems; take advantage of it. Customize your seminar to fit your group's specific needs.

- **Focus on specific topics.** Spend extra time on topics you really need and less time on topics you already know.

- **Provide an ideal experience.** For groups working on a project or looking to increase their knowledge of a particular topic, OSR's customized online seminars are ideal.

- **Save money.** The quote you receive from OSR includes everything you need. There are never additional charges for materials, shipping or expenses.

- **Save more money.** Attending an online seminar costs much less than sending several people to a public class.

- **Save hassles.** No need to arrange space or attendee lab systems.

