ISSUE VOLUME
01/22
MAY– JUNE
2016

The NT Insider

A publication of OSR Open Systems Resources, Inc.

```
; The commandments of x64 assembly:
;
;   1. Thou Shalt Not Write Inline Assembly
;   2. Thou Shalt Generate Unwind Data
;   3. Thou Shalt Comment Each Line of Assembly
;
include macamd64.inc

;
; External C function to read an article
;
; NTSTATUS
; TheNTInsiderReadSingleArticle(
;   PCHAR ArticleDescription,
;   ULONG_PTR PageNumber,
; );
;
EXTERN TheNTInsiderReadSingleArticle:PROC

.DATA
PeterPontificates      BYTE "COMPUTER SCIENCE EDUCATION? (YUP, STILL SUCKS)", 0
NewWaysToConnect       BYTE "INTRODUCTION TO SIMPLE PERIPHERAL BUS DEVICES AND DRIVERS", 0
TipsForUsingIoTargets  BYTE "A FEW RULES TO MAKE YOUR USE OF I/O TARGETS SIMPLE", 0
TodayInDriverSigning   BYTE "COLOR ME CONFUSED (STILL. AGAIN.)", 0
AnalystsPerspective    BYTE "MY DRIVER PASSES DRIVER VERIFIER! (OR DOES IT…)", 0
ByeByeCoInstallers     BYTE "SURPRISE? NEW VERSIONS OF WDF NO LONGER SUPPORTED DOWNLEVEL", 0

.CODE
NESTED_ENTRY TheNTInsiderReadEntireIssue, _TEXT

    save_reg rcx,  8h    ; Home RCX
    save_reg rdx, 10h    ; Home RDX
    save_reg r8,  18h    ; Home R8
    save_reg r9,  20h    ; Home R9

    alloc_stack 20h      ; Make home space for TheNTInsiderReadSingleArticle

    END_PROLOGUE         ; We are done manipulating the stack, so emit the
                         ; appropriate unwind stuff

    lea rcx, [PeterPontificates]    ; We're about to read the first article
    mov rdx, 4                      ; Put page number in RDX. I realize this comment
                                    ; isn't useful, but I'm supposed to comment every
                                    ; line...
    call TheNTInsiderReadSingleArticle ; Read the article!
    test eax, eax                   ; Returns an NTSTATUS, so check SF
    js Exit                         ; If it's set there's an error and we need to leave

    lea rcx, [NewWaysToConnect]     ; Time for the second article!
    mov rdx, 6                      ; Do what I did last time
    call TheNTInsiderReadSingleArticle ; Read the next article!
    test eax, eax                   ; Testin'...
    js Exit                         ; And jumpin'...

    lea rcx, [TipsForUsingIoTargts] ; Let's read another article!
    mov rdx, 8                      ; TODO: Learn to write a MASM loop...
    call TheNTInsiderReadSingleArticle ; Read it!
    test eax, eax                   ; This treats warnings as errors, but oh well...
    js Exit                         ; Yes, jump...

    lea rcx, [TodayInDriverSigning] ; Ditto
    mov rdx, 10                     ; Wait, why are page numbers 64-bit?
    call TheNTInsiderReadSingleArticle ; Read yet another article
    test eax, eax                   ; See previous comments
    js Exit                         ; A test engineer walks into a bar...

    lea rcx, [AnalystsPerspective]  ; More articles
    mov rdx, 12                     ; With more page numbers
    call TheNTInsiderReadSingleArticle ; Read it!!
    test eax, eax                   ; Why do we even let this fail?
    js Exit                         ; Leave if SF != 0...

    lea rcx, [ByeByeCoInstallers]   ; Last article
    mov rdx, 14                     ; Last page number
    call TheNTInsiderReadSingleArticle ; Read it!
    ; Fall through...
Exit:

    add rsp, 20h     ; Return the home space

    ret              ; Done!

NESTED_END TheNTInsiderReadEntireIssue, _TEXT

END
```

Inside:

- OSR Training
- Blog Updates
- Seminar Schedule

Published by
OSR Open Systems Resources, Inc.
105 Route 101A, Suite 19
Amherst, New Hampshire USA 03031
(v) +1.603.595.6500
(f) +1.603.595.6503

http://www.osr.com

Consulting Partners
W. Anthony Mason
Peter G. Viscarola

Executive Editor
Daniel D. Root

Contributing Editors
Scott J. Noone
OSR Associate Staff

Send Stuff To Us:
NtInsider@osr.com

Single Issue Price: $15.00

OSR Training
Let Us Help You!

You've got real commitments and project schedules to worry about. Making a decision to give up a week of your time to "learn up" is a big step. The value in that learning experience has many measurements. Here at OSR, it all starts with setting and meeting expectations of our attendees, and that's where I come in. My name is Debra Stitt, and I manage the team at OSR responsible for delivering a seminar experience that truly exceeds expectations.

What I love best about my position is the opportunity to help people every day. OSR may be well-known in the industry, but my team communicates with prospective attendees from all over the world, of varying technical backgrounds, and differing needs and goals. Determining "fit" is what it's all about for us, and that takes time and dedication. We're happy to spend the time necessary to do that to help you feel comfortable in a decision to choose OSR. Anything less is a disservice to you, and only hurts us both in the end.

Want to start a dialogue about a specific training need you have? Let's get started. Drop us a note with your interest and questions to seminars@osr.com.

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 Presentations:

Dulles/Sterling, VA
13-17 June

Seattle, WA
12-16 September

Get Social with OSR
Real -Time Updates

Follow us!

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

TH1, RS1, 1511, 14322—Happy Anniversary?
If you're having trouble following the lingo, code names, version numbers, and build numbers of the recent Windows releases you're not alone.
http://www.osr.com/blog/2016/05/13/th1-rs1-1511-14332-happy-anniversary/

Secrets of Using Win10 IoT Core on the RPI3 (and staying sane)
Let us save you some annoyance...
http://www.osr.com/blog/2016/04/15/secrets-using-win10-on-the-rpi-3/

Legacy File System Filters Blocked in Build 1607
THIS one shouldn't be a surprise, but undoubtedly someone will get bit...
http://www.osr.com/blog/2016/03/31/legacy-file-system-filters-blocked-build-1607/

More PI to Love...And Windows Supports It!
RPI3...
http://www.osr.com/blog/2016/02/29/pi-love-windows-supports/

Turning DbgPrint Statements into WPP Tracing
With inspiration from Chaucer...
http://www.osr.com/blog/2016/02/26/turning-dbgprint-into-wpp-tracing/

!pool Broken for Windows 10 Build 10586 Targets
Another public service announcement from OSR.
http://www.osr.com/blog/2016/01/14/pool-broken-windows-10-build-10586-targets/

Our Recommendations for Driver Signing—Windows 10 and Otherwise
And don't forget to read the related article in THIS newsletter
http://www.osr.com/blog/2015/12/29/recommendations-driver-signing-windows-10-otherwise/

Checked Kernel and HAL back in the WDK!
An oversight resolved...phew!
http://www.osr.com/blog/2015/12/14/checked-kernel-hal-back-wdk/

Sources/Dirs Converter? Gone from the Win10 V1511 WDK
Another one of those surprises that you'd hope for a "heads up" on...
http://www.osr.com/blog/2015/12/08/sources-dirs-coverter-gone/

Oops! VS 2015 Update 1 Breaks SDV
Ok, it HAS been fixed in Update 2...
http://www.osr.com/blog/2015/12/02/vs-2015-update-1-sdv/

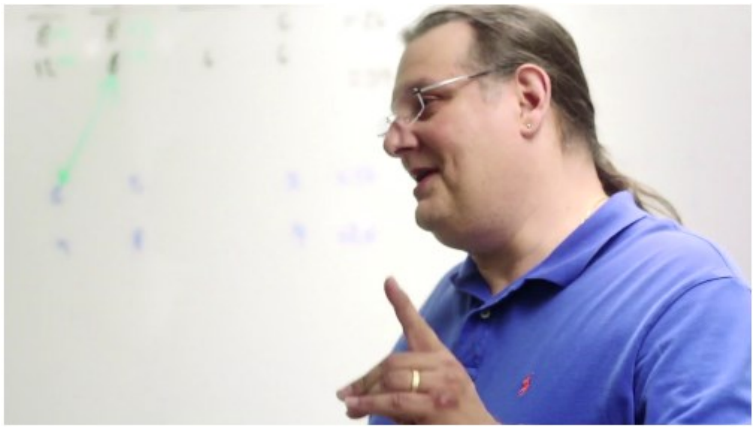
WdfWaitLockAcquire and Code Analysis: When SAL Goes Wrong
We live with them, but some shortcomings are worth pointing out.
http://www.osr.com/blog/2015/12/02/wdfwaitlockacquire-code-analysis-sal-goes-wrong/

Become More Knowledgeable... Instantly!

We email our friends when we've got something interesting to say. Join the list!

Send a blank email to join-osrhints@lists.osr.com and we'll add you to the list. We don't have THAT much to say. You'll probably get one or two emails a month.

Peter Pontificates
Computer Science Education? Yup. Still Sucks



I've been writing Peter Pontificates since 1996, when we made The NT Insider available (for free!) to anybody who was interested in the field of Windows System Software. Some of my Pontifications are intended to be funny – most of these weren't. Some attempt to predict the future – most of these work out wrong. A fair number comment on the state of the industry. When you look back on them years later, a lot of these Pontifications show the swings of the pendulum over time: Microsoft loves driver developers, Microsoft ignores driver developers, Microsoft sort of loves driver developers, Microsoft loves driver developers again.

What's really weird is that some situations in which our industry finds itself and on which I've pontificated over the years have not changed a bit. Back in 2002, I bemoaned the state of Computer Science education in the US. 2002. 14 years ago. George W. Bush was president of the United States (and I Pontificated at the time that it could never, ever, get worse… look who's running now!). Gerhard Schröder was German Chancellor. You could take a plane without having your lower intestinal tract inspected. Popular music was just starting to go downhill. Windows XP had shipped (but not the 64-bit version), and WS03 hadn't shipped yet.

It's against this backdrop that I wrote about how CS education – at least here in the States – was in dire need of fixing. I didn't think it could get worse. But, guess what? It has. Not only has it gotten worse, it's gotten much, much, worse. Recent CS grads now rarely even learn C. It's ridiculous. We see the evidence of this, every single day, in the posts on NTDEV. People write in with questions that demonstrate beyond a reasonable doubt that they do not know C, do not understand devices, and do not have any clue whatsoever about what an operating system is, does, or why you would want one.

Here, with modification, is exactly what I wrote in 2002:

I am totally depressed and disgusted by the state of computer science education here in the States. Once upon a time, it was impossible to graduate with a degree in computer science without knowing something about operating systems. In most "good" schools, you were required to take a minimum of two operating system classes, a compiler theory class, and a variety of languages including at least one assembler language.

These requirements ensured that new CS graduates were at least familiar with the fundamental principles of computer science down to the hardware level. You simply couldn't escape learning the basics of memory management, interrupts, ports, and registers. And every CS major had to be exposed to the principals of recursion, concurrency, and multi-threading. Learning about these topics provided you the basic grounding necessary to be a competent software engineer, regardless of the type of coding you eventually decided to pursue in your professional career.

Of course, even in those days not everybody was up to the challenge. If you wanted to work in the computer biz but basically not learn anything about computers, you could choose to major in something like "information technology" instead of computer science. Within this discipline you could learn really important things like how to write command procedures, execute SQL queries, and maybe even do backups. Hey, engineering isn't for everybody, right? And somebody's got to run those backups. At least nobody pretended these people were competent engineers.

What has me so seriously nauseated is that these days you can graduate from a reasonably well-respected university in the States and never learn an assembly language. Even worse, in many schools you can graduate with a CS degree without ever having taken an operating system theory class. Really. I am not kidding.

(CONTINUED ON PAGE 5)

Peter Pontificates... (Cont.)

(CONTINUED FROM PAGE 4)

I encounter recent CS grads all the time that have absolutely no idea – I mean none, zero, zip, nada – about how a virtual address is translated to a physical address. It might as well be by magic. Be clear about what I'm saying here: I'm not saying they don't know how virtual memory works in detail on some specific processor. I'm saying they don't understand the concept of virtual memory at all. You say "page fault" and they look back at you with a blank stare, as if you were reciting one of the Vedas. As for knowing the differences between running in kernel mode and user mode… forget about it. Interrupts? Ha! Memory mapped I/O? No way. Port space, device registers? No clue.

What's even scarier (like it could get scarier) is that these folks are equally ignorant of important fundamental concepts that can apply in user mode as well as in kernel mode, such as concurrency and multi-threading. "Multithreading…. That's something taken care of by the run-time library, isn't it?" Well, yes it is indeed! Here's your diploma. Please proceed directly to writing code in Java or TCL or something. When writing code in C#, please select the "threading model" of your choice from the list of radio buttons shown. Just whatever you do, stay way the f**k away from my kernel, OK?

I don't blame students for this mess. Hey, they don't know they're stupid. Students rely on the CS department to tell them what they need to study. And when universities are graduating kids who don't know that the words "register" and "port" have meanings other than those associated with food stamps and boats (respectively) then the schools are failing both their students and the industry. And up in Redmond, they actually wonder why so many drivers crash…

Is this problem peculiar to the States? I'm not sure, but there's evidence to suspect the situation is not nearly as hopeless everywhere. Have you noticed the increasing number of non-US trained engineers in the system software business? When was

(CONTINUED ON PAGE 27)

KERNEL DEBUGGING & CRASH ANALYSIS SEMINAR

I Tried !analyze-v...Now What?

You've seen our articles where we delve into analyses of various crash dumps or system hangs to determine root cause. Want to learn the tools and techniques yourself? Consider attendance at OSR's Kernel Debugging & Crash Analysis seminar. Here's what a recent OSR student said about his experience at this seminar:

"The instructor exhibited a very comprehensive knowledge of the material, added with an incredible ease in explaining a very complex subject. I highly recommend this course."

- Feedback from an attendee of THIS seminar

For more information, visit www.osr.com/seminars/kernel-debugging/, or contact an OSR seminar coordinator at seminars@osr.com

New Ways to Connect

Introduction to Simple Peripheral Bus (SPB) Devices and Drivers

Windows 8 quietly saw the introduction of many new Windows OS-level features. One of the most notable was support for devices connected via a Simple Peripheral Bus (SPB). SPBs are low-cost, low-power, low-speed buses that are most often used for connecting relatively simple peripherals such as sensors. Examples of SPB devices supported in Windows include I2C and SPI. Prior to Windows 8, these types of buses were restricted to use by the BIOS. But starting in Windows 8, support for these devices went mainstream.

There are several things that are interesting about SPB buses and about writing a driver for a device that's connected via an SPB bus. This article explores some of these topics.

SPB Buses – Topology and Enumeration

Like the SCSI, SATA, and USB buses, SPBs are protocol-based buses. That means there's a Controller that's responsible for getting requests on and off the bus using the appropriate protocol and according to the bus's specific rules. Devices that connect to protocol-based buses are called Client Devices.

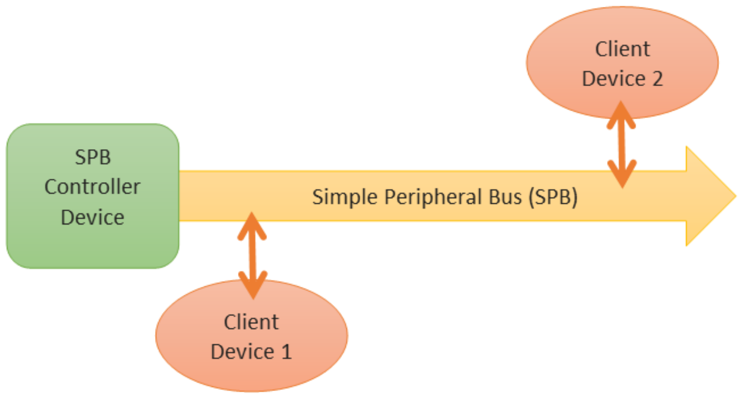


Figure 1— Controller, Bus and 2 Client Devices

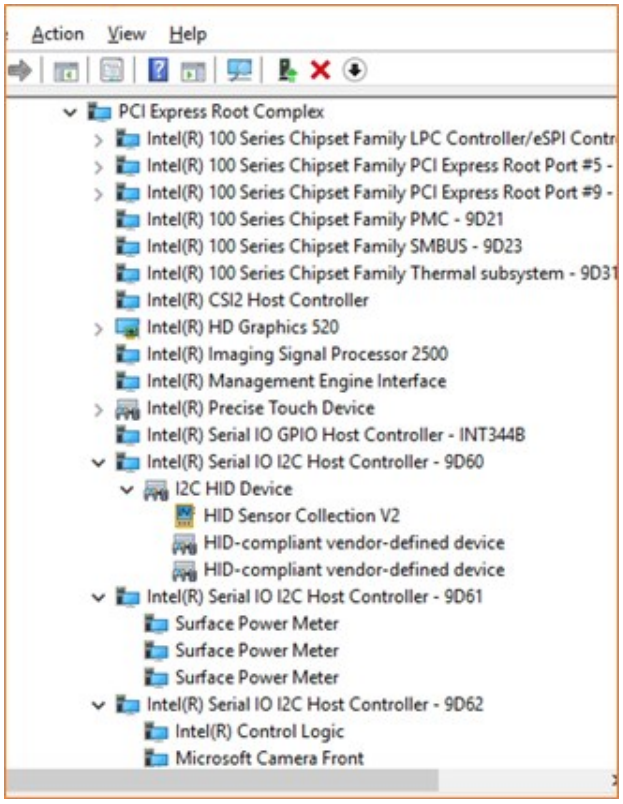
One point about these buses that causes some confusion is the way devices on an SPB bus are discovered and enumerated. SPB buses are not dynamically enumerable. That means that there's no way to discover which Client Devices, if any, are connected to a given SPB bus at run time. So, how **do** Client Devices on an SPB bus get discovered as part of the Windows PnP process? The answer is simple: The description of which Client Devices are attached to which specific SPB bus is supplied statically, in a table, as part of the ACPI BIOS. As a result, the Bus Driver that enumerates SPB Client Devices is ACPI, and not the SPB Controller Driver. Because the SPB Controller Driver is a standard backplane-bus type device it's enumerated by the PCI bus driver. You can see these points in **Figures 2 and 3** (P.7).

The ACPI table that contains this description is called the Differentiated System Descriptor Table or DSDT for short. The DSDT is usually provided in ROM and supplied by the system integrator, typically the OEM who builds the computer system. This works well because SPB Client Devices are usually permanently integrated into a system platform; That is, they're almost always soldered directly onto the system's main board. In the rare case that an SPB Client Device can be dynamically attached to a system (such as a specialized detachable keyboard that's connected via an I^{2}C bus), the information about the Client Device is still provided to the ACPI BIOS and the device will be enumerated by ACPI.

(CONTINUED ON PAGE 7)

SPB Devices and Drivers... (Cont.)

(CONTINUED FROM PAGE 6)



Action View Help

PCI Express Root Complex
Intel(R) 100 Series Chipset Family LPC Controller/eSPI Contr
Intel(R) 100 Series Chipset Family PCI Express Root Port #5 -
Intel(R) 100 Series Chipset Family PCI Express Root Port #9 -
Intel(R) 100 Series Chipset Family PMC - 9D21
Intel(R) 100 Series Chipset Family SMBUS - 9D23
Intel(R) 100 Series Chipset Family Thermal subsystem - 9D31
Intel(R) CSI2 Host Controller
Intel(R) HD Graphics 520
Intel(R) Imaging Signal Processor 2500
Intel(R) Management Engine Interface
Intel(R) Precise Touch Device
Intel(R) Serial IO GPIO Host Controller - INT344B
Intel(R) Serial IO I2C Host Controller - 9D60
I2C HID Device
HID Sensor Collection V2
HID-compliant vendor-defined device
HID-compliant vendor-defined device
Intel(R) Serial IO I2C Host Controller - 9D61
Surface Power Meter
Surface Power Meter
Surface Power Meter
Intel(R) Serial IO I2C Host Controller - 9D62
Intel(R) Control Logic
Microsoft Camera Front

Figure 2— Resources by Connection: Note the I^{2}C Controller and "Camera Front"

Common Uses

Windows supports both the SPI bus and the I^{2}C bus using the SPB model. The I^{2}C bus uses only 2 wires (plus power) for communication, making it an extremely simple interface. The SPI bus uses 4 wires (plus power). While initially these buses were primarily of interest to very small systems, such as smart phones and tablets, they have become very popular for interfacing "simple" devices on a wide variety of systems. In fact, Windows now includes I^{2}C as a standard method for connecting HID devices, and it is very popular among touchpad vendors. The Surface Pro 4 has more than a half dozen devices, ranging from cameras to power meters to a variety of HID devices that interface via the I^{2}C bus. You can see some of these devices in Figure 2 (which was captured on a Surface Pro 4).

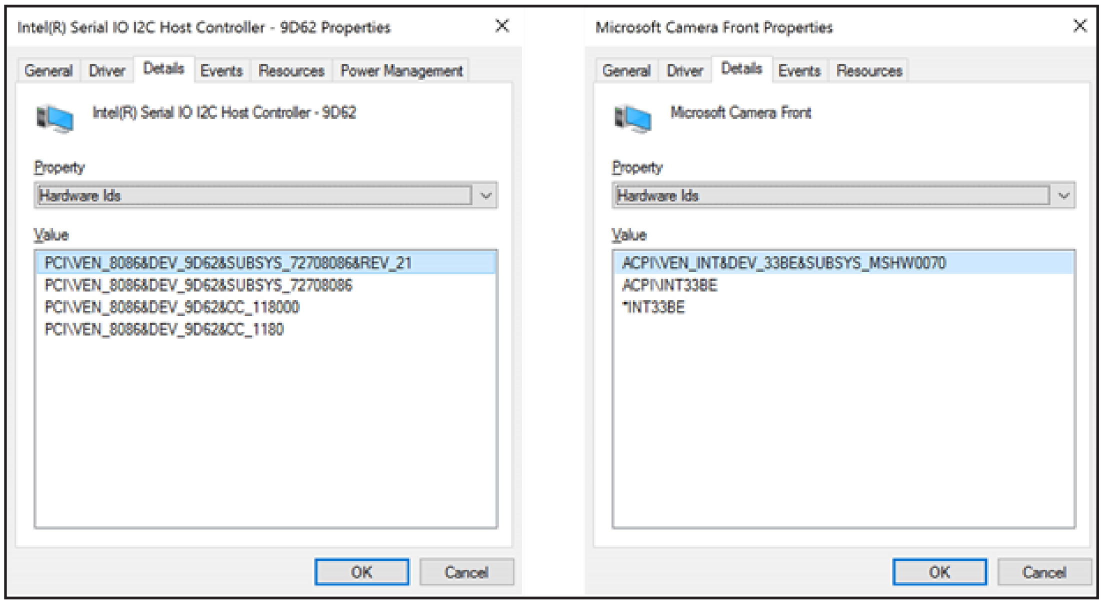
Writing SPB Drivers

As you can probably guess, there are two very different types of drivers that one could possibly write for SPB devices. There are drivers for the SPB Controller Device and drivers for SPB Client Devices.

Drivers for both types of SPB devices have special support in WDF that's provided by the SPB Class Extensions (SPBCx). SPBCx provides a standardized infrastructure that makes writing drivers for both categories of SPB devices much less difficult than it would be otherwise. Among other things, the SPBCx defines a set of I/O requests that a driver for a Client Device can send to the driver for a Controller Device to access their device. These standard I/O requests include specific rules for how read and write operations are processed by the driver for the Controller Device, as well as a standardized set of SPB-specific IOCTLs.

Aside from supporting the SPBCx, writing a driver for an SPB Controller Device is mostly like writing any driver for a device on a backplane architecture bus like PCI or PCIe. These drivers claim their hardware resources (such as registers, perhaps a DMA

(CONTINUED ON PAGE 22)



Intel(R) Serial IO I2C Host Controller - 9D62 Properties

General | Driver | Details | Events | Resources | Power Management

Intel(R) Serial IO I2C Host Controller - 9D62

Property: Hardware Ids

Value:
```
PCI\VEN_8086&DEV_9D62&SUBSYS_72708086&REV_21
PCI\VEN_8086&DEV_9D62&SUBSYS_72708086
PCI\VEN_8086&DEV_9D62&CC_118000
PCI\VEN_8086&DEV_9D62&CC_1180
```

OK Cancel

Microsoft Camera Front Properties

General | Driver | Details | Events | Resources

Microsoft Camera Front

Property: Hardware Ids

Value:
```
ACPI\VEN_INT&DEV_33BE&SUBSYS_MSHW0070
ACPI\INT33BE
*INT33BE
```

OK Cancel

Figure 3 — Hardware IDs for the Controller Device and Client Device (Camera) - Note the Bus Driver Names on Each

Tips for Using I/O Targets

A Few Rules to Make Your Use of I/O Targets Simple

In a recent article, I described three of the most common WDF errors that we, here at OSR, see in released drivers. One type of error that I described is the use, or rather **mis**use, of I/O Targets. The root cause of the problems in this category is that WDF allows you to use architecturally invalid combinations of activities involving I/O Targets without reporting an error. As a result, you can code-up your driver, test it, and even ship it… only to have your driver break later on when it encounters a slight change in its runtime environment.

In this article, I'll provide some guidelines for using WDF I/O Targets that – if you follow them – will ensure that you stay out of trouble with your use of **WdfRequestSend** and I/O Targets.

Background

In case you're not familiar with I/O Targets and **WdfRequestSend**, I'll give you a brief introduction. An I/O Target is a location to which you can send WDF Requests. There are two types of I/O Targets that are interesting in terms of our discussion: Local I/O Targets and Remote I/O Targets. The third type of I/O Target, named "special" I/O Targets, are related to USB (only) and are not particularly relevant to this article.

If your driver wants to send a Request to "the next device down" in its Device Stack, it retrieves the handle to its Local I/O Target by calling **WdfDeviceGetIoTarget**. If your driver wants to send a Request to a device in the system other than the device that's immediately below it in the Device Stack, it needs the handle to a Remote I/O Target that represents that device. To get this handle, the driver creates an empty I/O Target object using the function **WdfIoTargetCreate** and then opens that newly created I/O Target object using the function **WdfIoTargetOpen**. The target device can be described to the **WdfIoTargetOpen** function either by name or by providing a pointer to that device's existing native WDM Device Object.

When you send a Request to an I/O Target, you must supply the I/O parameters that will be sent with the Request to that I/O Target. These parameters include the I/O function code (Read, Write, DeviceControl, InternalDeviceControl), a description of the associated data buffer(s), and the offset on the device at which the operation should start (like, the offset from which to start reading or writing). The way you supply these parameters is by "formatting" the Request prior to sending it with **WdfRequestSend**. This format step is almost always an explicit step in setting up the Request to be sent, but in some very specific cases the formatting can be done implicitly. We'll talk about these cases when we discuss The Rules below.

That's a pretty quick description. If you need to know more, you should take our WDF seminar. Or search the web.

The Key to Understanding: Realizing Local and Remote Targets Are Very Different

The key thing to realize about using I/O Targets is that the operations you can perform, and how you handle the Request that you're going to send, is dependent on the type of I/O Target you're using. Once you understand the steps and allowed options for sending to a Local I/O Target and those for sending to a Remote I/O Target are different, you're on your way to writing safe, stable, correct code that will continue working even outside your test environment and into WDF versions in the future.

In the following section I'll describe some of the basic rules. In this article, I'm going to stick to the most common design patterns and the major rules. As a result, **I'm going to consciously ignore some of the less-used design patterns and some of the things that are possible but are rarely done**. So, if you're an experienced WDF developer or you're a member of the WDF development

(CONTINUED ON PAGE 9)

A Few Rules... (Cont.)

(CONTINUED FROM PAGE 8)

team, don't get all upset that I haven't described your favorite special case. The rules I describe here err on the side of being conservative. As a result, I can confidently say that if you stick to these rules below, you'll never go wrong when using I/O Targets.

The Rules for Sending to a Remote I/O Target

Recall that you can open a Remote I/O Target by name or by providing a pointer to an existing native WDM Device Object.

```
status = WdfRequestRetrieveOutputMemory(Request, &outputMemory);

if (!NT_SUCCESS(status)) {
    WdfRequestComplete(Request, status);
    return;
}

//
// Step 1: Format with reference to the specific I/O Target to which
// we'll be sending the Request.
//
status = WdfIoTargetFormatRequestForIoctl(devContext->Target,
                                          Request,
                                          IoControlCode,
                                          NULL,
                                          NULL,
                                          outputMemory,
                                          NULL);

if (!NT_SUCCESS(status)) {
    WdfRequestComplete(Request, status);
    return;
}

//
// Set a completion callback... we'll be sending the Request async.
//
WdfRequestSetCompletionRoutine(Request,
                               MyDriverRequestCompletionRoutine,
                               NULL);

//
// Step 2: Send the Request to the Remote I/O Target
//
ret = WdfRequestSend(Request,
                     devContext->Target,
                     WDF_NO_SEND_OPTIONS);

if (ret == FALSE) {
    status = WdfRequestGetStatus (Request);
    WdfRequestComplete(Request, status);
}
```

FIGURE 1 – Formatting a Request for Remote I/O Target

The primary rule is: ***If you're sending a Request to a Remote I/O Target, you must format the Request for that specific I/O Target*** before sending it. That means you must call one of the formatting functions that starts with **WdfIoTarget**, such as **WdfIoTarget FormatRequestForXxxx** (where Xxxx is Read, Write, Ioctl, and InternalIoctl). There is no other way that's legal. Note, specifically, that it is not correct (or even supported) to call **WdfRequestFormatRequestUsing CurrentType** before you send a Request to a Remote I/O Target. Judging by a lot of the code I've read, this will come as a surprise to a lot of people. You have to use a method that's specific to the I/O Target to which you'll be sending the Request. Hence, the method you use must start with **WdfIoTarget**.

When you send the Request, you may send the Request to the Remote I/O Target either Synchronously or Asynchronously with a callback. There are some very narrow, very limited cases where it's possible to send a Request to a Remote I/O Target using **_SEND_AND_FORGET** but these cases are so limited that they're not worth discussing. So, forget **_SEND_AND_FORGET** for Remote I/O Targets. Just remember that if you're sending a Request to a Remote I/O Target, you must send it either Synchronously or Asynchronously with a callback.

Pretty simple, right? Right: Format the Request using **WdfIoTargetFormatRequestForXxxx**, specify a WDF_REQUEST_SEND_OPTIONS structure specifying either synchronous processing or asynchronous processing with a callback, call **WdfRequestSend**, and you're done. You can see this in code in **Figure 1**.

Note that you can use this pattern for any WDF Request that you're sending to a Remote I/O Target. In other words, you can use it to send Requests that you receive from a Queue (so called Queue Presented Requests) or Requests that you create in your driver by calling **WdfRequestCreate**.

Oh, one more thing: If you specify synchronous processing, be absolutely sure you have specified an **ExecutionLevel** constraint on your WDFDEVICE or WDFQUEUE as **WdfExecutionLevelPassive**. This is the only time you can use synchronous processing.

The Rules for Sending to a Local I/O Target

The first thing to understand about sending Requests to Local I/O Targets is that you can always use the same pattern that you use for sending to Remote I/O Targets. That is, you can format the Request using **WdfIoTargetFormatRequestForXxxx** and then send

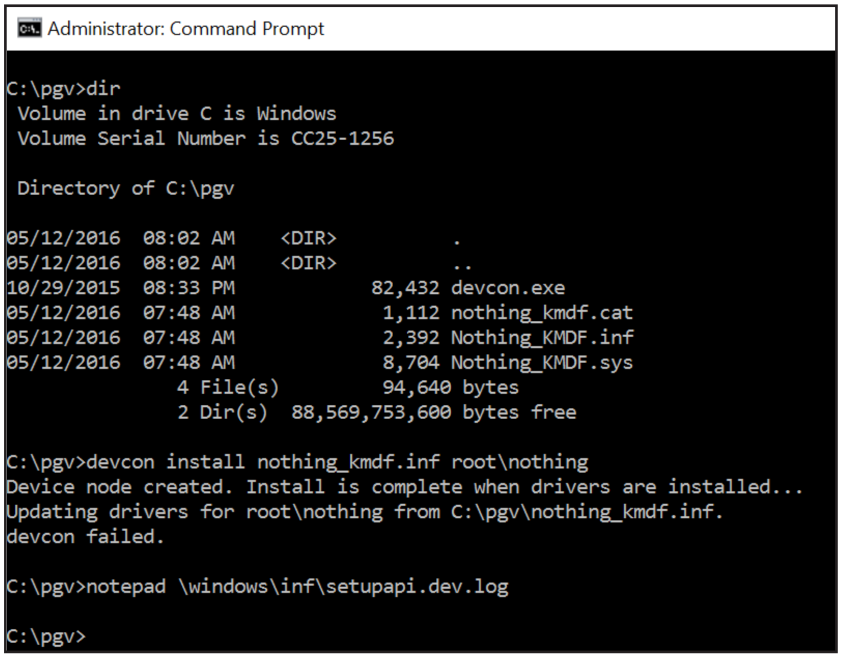
(CONTINUED ON PAGE 20)

Today in Driver Signing
Color Me Confused (Still. Again.)

If you've been following the events of the past couple of years regarding driver signing, you'll know that there's been a lot of stuff that's unclear. For those of you who don't live and breathe Windows driver policies, let me summarize for you: Prior to the introduction of Win10 (TH1... can't remember which version, code name, or build of Windows is which? Us either... Check out this handy blog post), Microsoft announced that Windows 10 would not allow installation of drivers unless the driver was signed via the SysDev portal (that is, signed by Microsoft, though this will not require the driver to pass the HLK tests). With the help of Microsoft PM James Murray, we tried to flesh out the details of this policy. Throughout the Windows 10 release cycle, this policy changed. The current policy is murky, even with the clarifying statements that have been issued, but seems to suggest that in Win10 TH1 and TH2, you can still install drivers using the long-standing, traditional method of cross-signing. This is supported by lots of real-world experience. The driver development community breathed a collective sigh of relief.

Fast forward to recent times. The pending release of Windows 10 Anniversary Edition (code name Redstone 1, RS1 for short) scheduled for release in July of 2016, once again raises the question of "will the long-standing driver signing policy change"? Community experience as described in this NTDEV thread seems to indicate that the requirement that newly installed kernel-mode drivers be signed by the SysDev portal will indeed start to be enforced in RS1. There are also reports that this will only be the case if Secure Boot is enabled on the platform. Various people have tested this.

But, for you dear reader, I decided to test things myself. My goal was to be able to make some definitive statements about what was going on. The results were not what I expected.


```
Administrator: Command Prompt

C:\pgv>dir
 Volume in drive C is Windows
 Volume Serial Number is CC25-1256

 Directory of C:\pgv

05/12/2016  08:02 AM    <DIR>          .
05/12/2016  08:02 AM    <DIR>          ..
10/29/2015  08:33 PM            82,432 devcon.exe
05/12/2016  07:48 AM             1,112 nothing_kmdf.cat
05/12/2016  07:48 AM             2,392 Nothing_KMDF.inf
05/12/2016  07:48 AM             8,704 Nothing_KMDF.sys
               4 File(s)         94,640 bytes
               2 Dir(s)  88,569,753,600 bytes free

C:\pgv>devcon install nothing_kmdf.inf root\nothing
Device node created. Install is complete when drivers are installed...
Updating drivers for root\nothing from C:\pgv\nothing_kmdf.inf.
devcon failed.

C:\pgv>notepad \windows\inf\setupapi.dev.log

C:\pgv>
```

Figure 1 — No Signature = No Install. Just What You'd Expect.

Can You Install a Cross-Signed Driver on RS1? Yes.

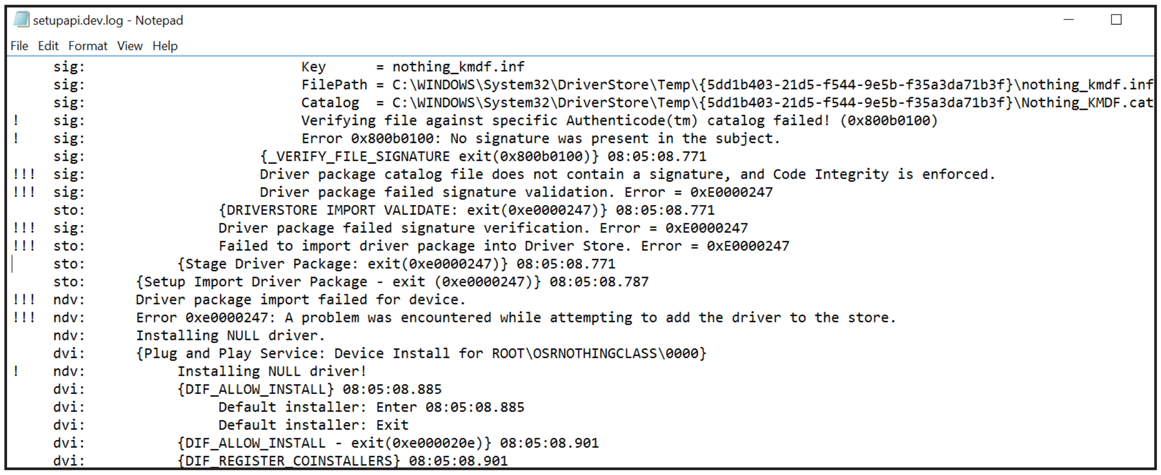
I started my testing on a spare Surface Pro 1 we had kicking around. Because the Surface Pro 1 isn't super-happy about booting random USB drives (no, I don't know why... but I don't know anybody who's managed to get it to work), I reimaged the system using the standard Windows 8 restore disk. I then downloaded Build 14295 from the Windows Insider web site and attempted a clean-install from the ISO (running the install from the Windows 8 system). I then waited for the "Fast Ring" to push me Build 14332, which dates from 22 April 2016. Note: Secure Boot was enabled on the system. The debugger was not attached, and kernel debugging was not enabled. Test Signing was not enabled. We're talking a standard system install of a "Fast Ring" system here. Nothing cute.

Just to make sure things were sane, I started by trying to install my software-only driver. I didn't sign anything. The results were as expected, and shown in **Figure 1**. The driver wouldn't install.

(CONTINUED ON PAGE 11)

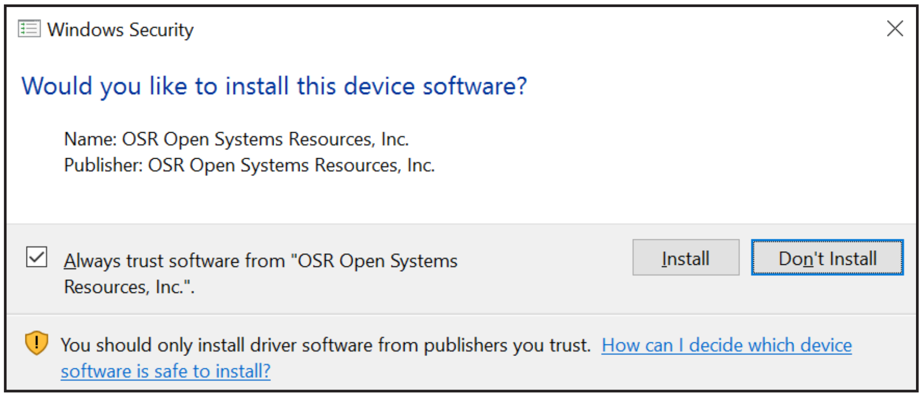
Color Me Confused... (Cont.)

(CONTINUED FROM PAGE 10)



```
     sig:                          Key      = nothing_kmdf.inf
     sig:                          FilePath = C:\WINDOWS\System32\DriverStore\Temp\{5dd1b403-21d5-f544-9e5b-f35a3da71b3f}\nothing_kmdf.inf
     sig:                          Catalog  = C:\WINDOWS\System32\DriverStore\Temp\{5dd1b403-21d5-f544-9e5b-f35a3da71b3f}\Nothing_KMDF.cat
!    sig:                          Verifying file against specific Authenticode(tm) catalog failed! (0x800b0100)
!    sig:                          Error 0x800b0100: No signature was present in the subject.
     sig:                     {_VERIFY_FILE_SIGNATURE exit(0x800b0100)} 08:05:08.771
!!!  sig:                     Driver package catalog file does not contain a signature, and Code Integrity is enforced.
!!!  sig:                     Driver package failed signature validation. Error = 0xE0000247
     sto:                {DRIVERSTORE IMPORT VALIDATE: exit(0xe0000247)} 08:05:08.771
!!!  sig:                Driver package failed signature verification. Error = 0xE0000247
!!!  sto:                Failed to import driver package into Driver Store. Error = 0xE0000247
     sto:           {Stage Driver Package: exit(0xe0000247)} 08:05:08.771
     sto:      {Setup Import Driver Package - exit (0xe0000247)} 08:05:08.787
!!!  ndv:      Driver package import failed for device.
!!!  ndv:      Error 0xe0000247: A problem was encountered while attempting to add the driver to the store.
     ndv:      Installing NULL driver.
     dvi:      {Plug and Play Service: Device Install for ROOT\OSRNOTHINGCLASS\0000}
!    ndv:           Installing NULL driver!
     dvi:           {DIF_ALLOW_INSTALL} 08:05:08.885
     dvi:                Default installer: Enter 08:05:08.885
     dvi:                Default installer: Exit
     dvi:           {DIF_ALLOW_INSTALL - exit(0xe000020e)} 08:05:08.901
     dvi:           {DIF_REGISTER_COINSTALLERS} 08:05:08.901
```

Figure 2 — No, You Can't Install an Unsigned Driver on 64-bit Windows. Duh!



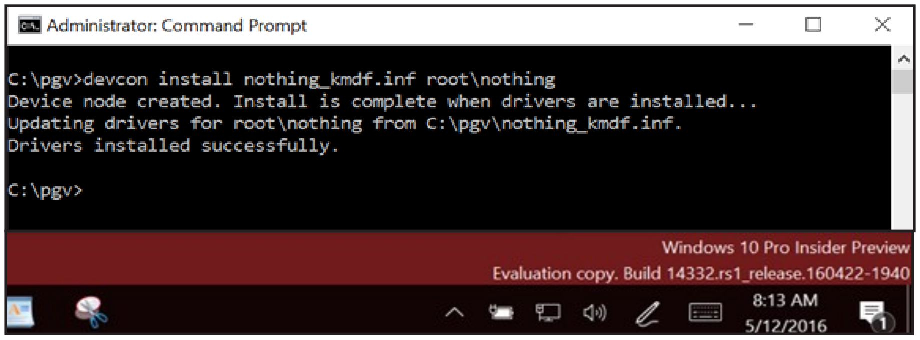
Would you like to install this device software?

Name: OSR Open Systems Resources, Inc.
Publisher: OSR Open Systems Resources, Inc.

☑ Always trust software from "OSR Open Systems Resources, Inc.".

You should only install driver software from publishers you trust. How can I decide which device software is safe to install?

Figure 3 — Yes, I want to install this "device software"! But...


Administrator: Command Prompt

```
C:\pgv>devcon install nothing_kmdf.inf root\nothing
Device node created. Install is complete when drivers are installed...
Updating drivers for root\nothing from C:\pgv\nothing_kmdf.inf.
Drivers installed successfully.

C:\pgv>
```

Windows 10 Pro Insider Preview
Evaluation copy. Build 14332.rs1_release.160422-1940
8:13 AM
5/12/2016

Figure 4 — Signed and Cross Signed; Installs Just Fine on 14332, Thank You.

The setupapi.dev.log file (**Figure 2**) had the usual error messages stating "Driver package catalog file does not contain a signature, and Code Integrity is enforced" and "Driver package failed signature validation" -- All good so far, right?

Then, I signed the driver executable and catalog files in the “traditional” way using the appropriate cross-signing certificate with one of OSR's Release Signing certificates (for the record, this is a Verisign issued Class 3 Code Signing certificate issued in March of 2016 using SHA-256, but without Extended Validation). I copied the signed package to the RS1 system and attempted the install. I was greeted with the familiar message box shown in **Figure 3**. And the installation worked perfectly (**Figure 4**).

Whoa! Wasn’t this supposed to **fail**? Isn’t this the whole point we’re trying to demonstrate? Isn’t RS1 supposed to require newly installed drivers to be signed by Microsoft? Hmmm... well, in this test, not so much.

(CONTINUED ON PAGE 16)

Analyst's Perspective
My Driver Passes Driver Verifier! (Or does it…)

A fundamental complexity in Windows kernel mode development is that the execution environment comes from a different era in software development. Basically, the idea is this: if you're writing kernel mode code, then you must know what you're doing. If you know what you're doing, then we don't need to validate your function parameters and therefore we can shave off some CPU cycles. We also don't need to bother validating the execution environment at all because, you know, everyone knows what they're doing.

If you don't know what you're doing, then you're stupid. If you don't pass valid arguments, you get what you deserve. If you don't understand the rules of the execution environment, then kernel mode software development must be too hard for you.

The problem with all of this of course is that this approach doesn't really scale. Without proper validation, you can easily crash the system by calling a function with an invalid argument. Even worse, the system might **not** crash but instead subtly corrupt an internal structure or return an invalid result. Also, you need great documentation for people to, you know, actually **learn** what the rules are for the environment.

Documentation issues aside, Windows 2000 introduced Runtime Driver Verifier to address the issue of insufficient runtime validation of arguments and execution environment. This allows us to put the OS into a special mode where the drivers aren't trusted and we can gain the benefits of OS level validation. With each iteration of Windows, Verifier has become more and more powerful and maintains its title as the single greatest gift that the Universe has bestowed upon driver developers. Passing Verifier is the **minimum** requirement for professional software development in the Windows operating system. If you're not running your driver under Verifier, you have failed. Seriously.

I was recently talking about Verifier with an IT administrator for a large organization. He mentioned that he had a lot of systems crashing and went around enabling Verifier for various third party drivers on the systems hoping to find the culprit. The systems started crashing immediately and directly pointing to a bug in a third party driver. After bringing the crash up with the vendor, their response was, "shut Verifier off, we don't test with that." This is so wrong that I'm close to publicly shaming the company, I'm just not sure what they're thinking. My suggestion to the IT admin was to beat the company harder and, if they won't listen, find a replacement product.

"Fools!" you say, "I use Verifier **all the time**. I am safe!" However, you might be missing something critical in your testing: just enabling Verifier for **your driver only** is hardly ever sufficient. Do you have a KMDF driver? A FltMgr filter? An NDIS driver? StorPort miniport? For any of these, you really need to enable Verifier on both your driver **and** the wrapper library!

(CONTINUED ON PAGE 13)

My Driver Passes... (Cont.)

(CONTINUED FROM PAGE 12)

The problem is that Verifier is validating calls into the operating system. For the above drivers, *your* driver isn't calling into the operating system, the library is. For example, if you're KMDF driver calls WdfDeviceCreate, it's the **Framework** that calls ExAllocatePool to allocate your WDFDEVICE and your Device Context. The buffers allocated in this case won't be subject to validation by Driver Verifier unless you have explicitly enabled Driver Verifier for the Framework. If you only enable Driver Verifier for your driver, the only pool validation that you get will be for calls that your driver makes directly to ExAllocatePool (**Figure 1**).

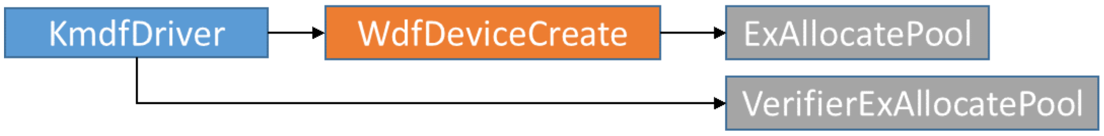


Figure 1 — No Special Pool!

So, the rule is, when you enable Driver Verifier for your driver, always also enable Driver Verifier for the wrapper/library that your driver uses (**Table 1**).

Another option that people frequently miss: you can also enable Verifier on the NTOS Kernel image! This means that allocations made by the OS itself (e.g. File Objects) will also be subject to Verifier's checking. This results in a unique form of parameter validation that you might not catch otherwise.

You Write This Type of Driver	Enable Verifier on Your Driver, AND ALSO
KMDF	Wdf01000.sys
NDIS	ndis.SYS
File System Filter	fltMgr.SYS
StorPort	Storport.sys

Table 1 — Enable Verifier on your Driver's Executable and the Wrapper

Ultimately, the lesson is that more Verifier is a good thing so make sure you enable it early, often, and for any driver that your driver touches. Of course, the downside to Verifier is that the system behaves differently when Verifier is enabled, thus you're not actually testing the real customer environment. So, unless you're going to make all your customers turn on Verifier as part of install, make sure you also test **without** Verifier enabled as part of your QA. See the sidebar, *Still Want More Validation...* below for another helpful tip.

Follow us!

Still Want More Validation? Be Sure to Try WDFVerifier and the Checked Build

If you're writing a WDF driver, you definitely want to also enable WDF Verifier. See the topic Using KMDF Verifier in the WDK Documentation (Google for it when the provided link breaks, as it will).

Whether you're writing a WDF driver or not, there's still a lot of additional checking of which you can take advantage. For example, at some point you should always test with the checked build of Windows, the checked build of any wrapper/library components used by your driver, and the checked build of any driver(s) with which your driver interacts. The checked OS image and HAL are distributed as part of the WDK. You can find the documentation under Downloading a Checked Build of Windows in MSDN. In addition to these, download a complete checked build of Windows (assuming you can find it; they're getting harder and harder to find with each OS release). You can either choose to install the complete checked build, or you can extract (with some work) just the checked images for the wrapper/library that your driver uses, plus the checked executables for any drivers with which your driver interacts. For example, if you're writing a file system filter driver, you'll want to use the checked build of Filter Manager (fltMgr.sys) plus the checked builds of the file systems that you filter. Our experience is that this can be very helpful in identifying potential problems.

Bye-Bye CoInstallers!
Surprise? New Versions of WDF No Longer Supported Downlevel

Maybe you're like me, and you missed the memo that must have been circulated two years ago. But I just learned, via a thread on NTDEV started by long-time driver developer Ed Dekker that KMDF versions 1.13, 1.15, and 1.17 cannot be used on older versions of Windows.

Throughout the history of WDF, devs have had the option of writing drivers to whichever version of the WDF Framework that they chose. They could then ship their driver along with that version of the Framework packaged in a "CoInstaller DLL." If the version of WDF used by the driver was newer than the version that was available on the system on which the driver was being installed, the CoInstaller would update the version on the system.

What was nice about this is that it would allow a WDF driver to use certain features that were available in a newer version of the Framework even on older versions of Windows. What was bad about it is that (a) Updating the version of WDF required a reboot, and (b) the system on which the driver was being installed had to support co-installers. Starting with Windows 8, not every system that runs Windows supports installing drivers using co-installers (think, IoT Core for one example).

However, as community leader Tim Roberts points out in the previously referenced NTDEV thread, there's a table in MSDN that very clearly notes that:

- KMDF V1.13, which was released on Windows 8.1, will only run on Windows 8.1 or later;
- KMDF V1.15, which was released on Windows 10 (TH1), will only run on Windows 10 or later;
- KMDF V1.17, which was released on Windows 10 (TH2), will only run on Windows 10 TH2 or later.

Wait! Don't go nuts. You can still go UP level without any problem. So, for example, you can write your driver using KMDF V1.11 and install it without incident on Windows 8.1 on which V1.13 is installed. This is, of course, what's **really** important. The ability to go down-level was always merely a convenience.

And if you want to look at this more from the "glass half **full**" perspective, the good news is that if you target your driver to Windows 8.1 or later you don't have to ship-around the co-installer in your driver package anymore.

So, given that at least one person here at OSR is writing a WDF driver every day of every week, why didn't we notice this change until Mr. Dekker happened to ask his question on NTDEV? Well, except for the architectural concept, it's not clear how much this matters in the real world. In practice, we've built drivers that target every reasonably old OS that KMDF supports (defined as

(CONTINUED ON PAGE 15)

WE KNOW WHAT WE 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 us 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 we can help. If we can't help you, we'll tell you that, too.

Contact: sales@osr.com

Surprise?... (Cont.)

(CONTINUED FROM PAGE 14)

Windows XP and later) and used KMDF V1.11 with a co-installer, or else we've built drivers that target Windows 10 or later and used KMDF V1.15 without a co-installer.

Not to mention, as Mr. Roberts so succinctly put it in that same NTDEV thread:

```
I haven't seen anything in KMDF beyond 1.11 that compels me to switch.
```

Yeah. There's that, too.

So now you know! Truth be told, I'm kinda happy to get rid of those co-installers anyways.

Follow us!

Color Me Confused... (Cont.)

(CONTINUED FROM PAGE 11)

After doing this test, I remembered that there was some speculation that if the system was an upgrade from a previous version, it would load and install drivers that were signed in the "traditional" way. To test this out, I downloaded the latest Insider Program ISO (build 332) and installed it to a brand-new VMware virtual machine. That should fail, right? A new install of the latest RS1 build? See **Figure 5**.

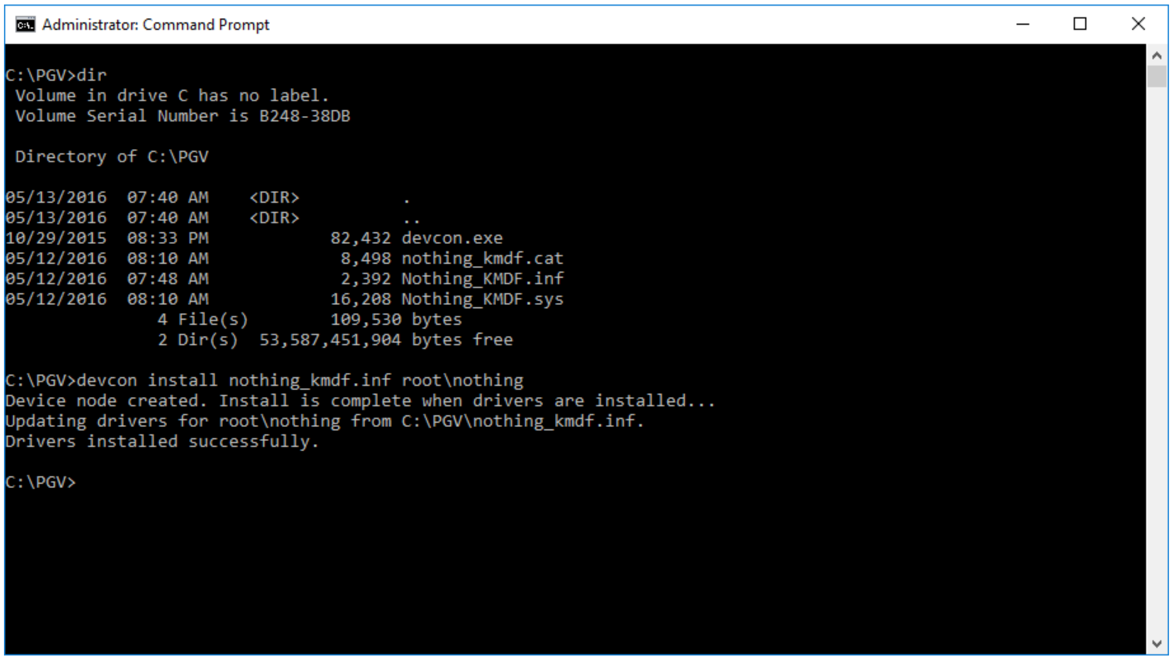
Sigh. It still works. So, let's summarize:

- You can't install an unsigned x64 driver. Duh.
- Upgraded RS1, Build 14332: Traditionally signed (and cross-signed) driver installs fine.
- New install of RS1, Build 14332: Traditionally signed (and cross-signed) driver installs fine.

So… at least based on tests of the latest Insider builds available, no unique signing procedure is necessary to get drivers to load on Windows 10 Anniversary Update.

However, according to that same thread on NTDEV thread that I referred to previously, the requirement **may** kick-in if the system is both newly installed **and** booted in "Secure Boot" mode. Unfortunately, we weren't able to try this. Why? Because we couldn't find a system at OSR that was both (a) capable of booting in Secure Boot Mode, and (b) able to be clean-installed. (And here's an interesting digression: Have you ever tried to do a clean install on a Surface Pro 1? No? Let me just say that you should give it try sometime when you have a few days to waste. In short, it's not happening no matter what you do. Upgrade? Yes. Actual clean install? No. It's maddening. Why can't the damn system boot a USB drive with the install kit on it, like a normal system? Arrrgh!)

(CONTINUED ON PAGE 17)



```
C:\PGV>dir
 Volume in drive C has no label.
 Volume Serial Number is B248-38DB

 Directory of C:\PGV

05/13/2016  07:40 AM    <DIR>          .
05/13/2016  07:40 AM    <DIR>          ..
10/29/2015  08:33 PM            82,432 devcon.exe
05/12/2016  08:10 AM             8,498 nothing_kmdf.cat
05/12/2016  07:48 AM             2,392 Nothing_KMDF.inf
05/12/2016  08:10 AM            16,208 Nothing_KMDF.sys
               4 File(s)        109,530 bytes
               2 Dir(s)  53,587,451,904 bytes free

C:\PGV>devcon install nothing_kmdf.inf root\nothing
Device node created. Install is complete when drivers are installed...
Updating drivers for root\nothing from C:\PGV\nothing_kmdf.inf.
Drivers installed successfully.

C:\PGV>
```

Figure 5 — Install of "Traditionally" Signed Drivers on New RS1 System. Yup, Still Works.

Color Me Confused... (Cont.)

(CONTINUED FROM PAGE 16)

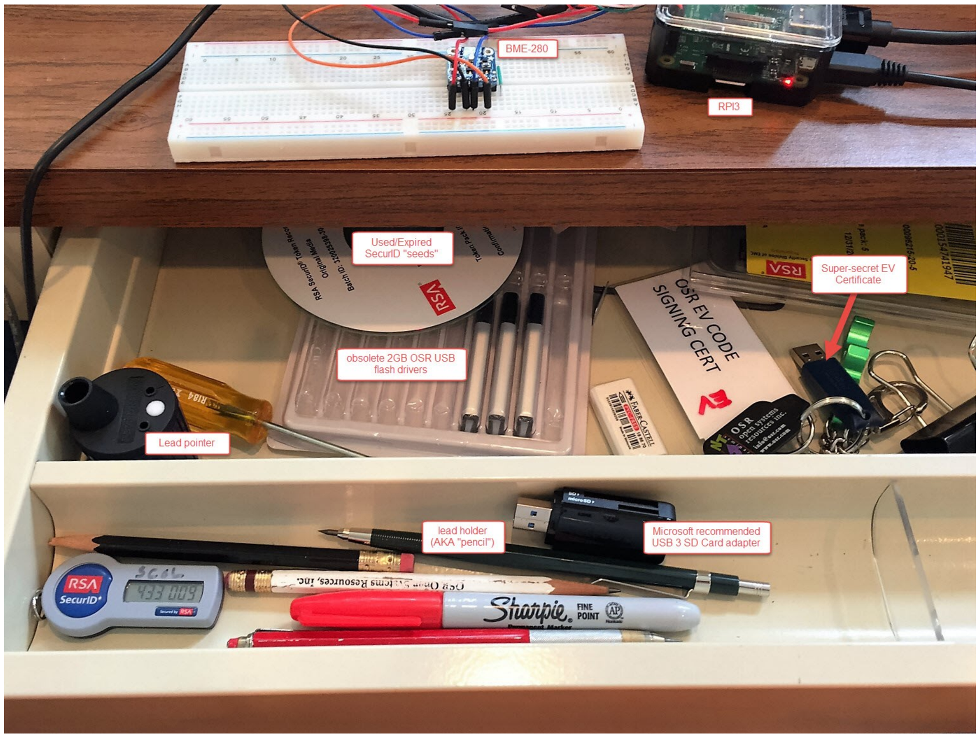

BME-280
RPI3
Used/Expired SecurID "seeds"
Super-secret EV Certificate
obsolete 2GB OSR USB flash drivers
Lead pointer
lead holder (AKA "pencil")
Microsoft recommended USB 3 SD Card adapter

Figure 6 – Apparently Ease of Access Wins Over Security...

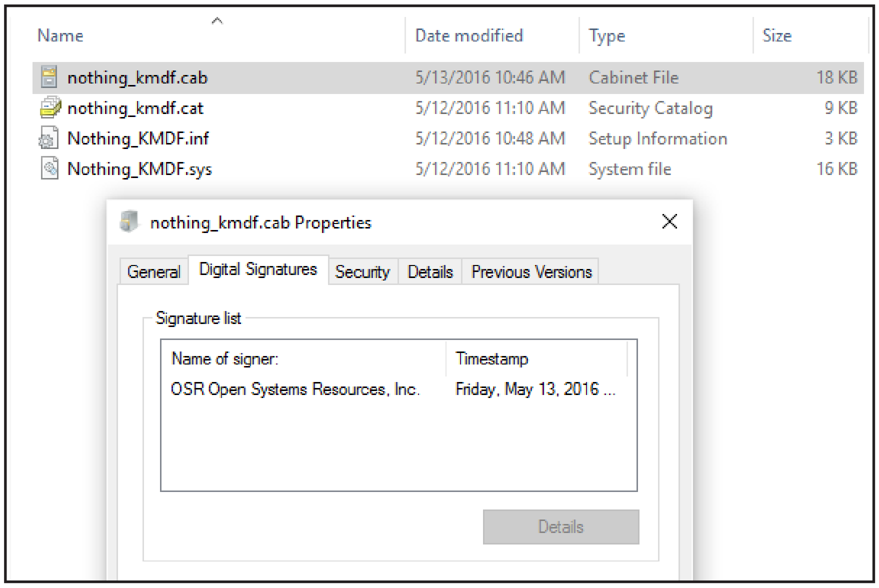
Does Attestation Signing Work? Is it a PITA?

While we were wasting our time, we **did** have one more thing that we wanted to try. We wanted to walk through the Attestation Signing process to see if it's as easy as it should be… and to see on which systems you could install Attestation Signed drivers.

The Attestation Signing process is reasonably well described on MSDN, in a Dev Center post by Don Marshall. We'll walk you through the process and try to make it a bit easier.

Before you can upload your drivers to the Microsoft SysDev portal for Attestation Signing, you have to take your driver package and bundle it into a CAB file. Right. Not a ZIP file. A CAB file. Microsoft provides the lovely MakeCab utility, which works but is the very definition of Royal Pain In The Ass. For example, just to put stuff into a CAB file using MakeCab, you have to create a separate DDF file that describes what you want MakeCab to do. Screw that, I say. I recommend that you use a very nice little GUI utility named IZArc (that's an "I" not an "L"), that you can download from here. No, I have no relationship to and don't know anything about the author. But his utility worked swell for me.

So, first… sign (and cross-sign) everything you can in your driver package. Sign the CAT file, sign the SYS file (or files), sign any DLLs, applications… just sign everything. It's yours, you wrote it, you should sign it. Note that the Attestation Signing signature will overwrite (**not** add to) the signature on your CAT file (**very** annoying), but will actually be added to the signature(s) you provide in your SYS file.



Name	Date modified	Type	Size
nothing_kmdf.cab	5/13/2016 10:46 AM	Cabinet File	18 KB
nothing_kmdf.cat	5/12/2016 11:10 AM	Security Catalog	9 KB
Nothing_KMDF.inf	5/12/2016 10:48 AM	Setup Information	3 KB
Nothing_KMDF.sys	5/12/2016 11:10 AM	System file	16 KB

nothing_kmdf.cab Properties

Name of signer:	Timestamp
OSR Open Systems Resources, Inc.	Friday, May 13, 2016 ...

Figure 7 — Signed CAB Ready for Upload

So, use IZArc to stuff everything in your driver package into a CAB file. Be careful in formatting the CAB file: Put the driver package in its own directory. Do not put any files in the root directory of the CAB file.

Recall that in order to be able to submit your CAB file to SysDev, you need to sign it using your super-special SysDev EV Code Signing Certificate. So, off I went to OSR's vault to retrieve our EV Code Signing Cert hardware token from its ultra-secure location (**Figure 6**). I signed the CAB file I created using this certificate. You can see what I put into the CAB file, the CAB file itself, and the signature info for the CAB file, in **Figure 7**.

(CONTINUED ON PAGE 18)

Color Me Confused... (Cont.)

(CONTINUED FROM PAGE 17)

So, to begin the Attestation Signing process itself, it's off to the Microsoft SysDev Portal at https://sysdev.microsoft.com. Oh… by the way… we couldn't get IE 11 to work properly with SysDev. We had to use either Chrome or Edge. So, Chrome it was. Note that we had previously arranged for SysDev access, signed all the necessary agreements, and verified our EV Code Signing Certificate.

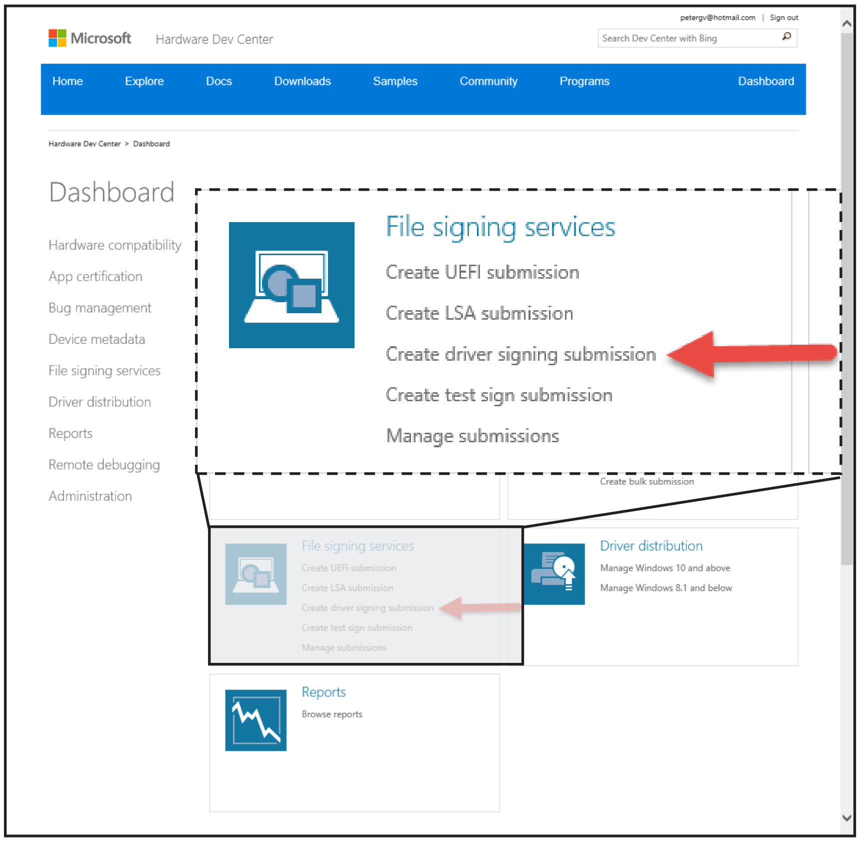
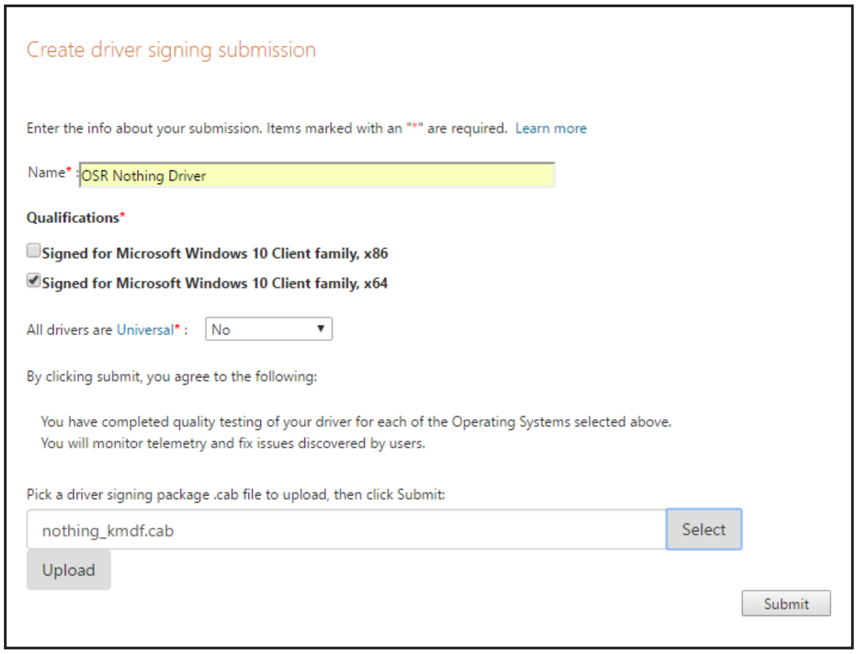


Figure 8 — Login to SysDev ... Click Where Indicated

When you first login to SysDev, you're greeted with the interface shown in **Figure 8**.

To upload a driver to sign by "Attestation", click the Create Driver Signing Submission option indicated by the red arrow in Figure 8.

You'll next be taken to the File Signing Services page where you can fill out the "Create driver signing submission" form shown in **Figure 9**. It's pretty simple to fill out. Nothing tricky here at all. Click "Submit" and you're on your way.


Create driver signing submission

Enter the info about your submission. Items marked with an "*" are required. Learn more

Name* : OSR Nothing Driver

Qualifications*

☐ Signed for Microsoft Windows 10 Client family, x86

☑ Signed for Microsoft Windows 10 Client family, x64

All drivers are Universal* : No

By clicking submit, you agree to the following:

You have completed quality testing of your driver for each of the Operating Systems selected above.
You will monitor telemetry and fix issues discovered by users.

Pick a driver signing package .cab file to upload, then click Submit:

nothing_kmdf.cab

Figure 9 — Fill it In, and Click Submit

You'll get a submission ID and you can monitor the progress of your submission via the "Manage submissions" page. It took me less than 30 minutes to get an email from "sysdev@microsoft.com" telling me that my driver had been signed and was ready for download (**Figure 10, next page**).

From there, all I had to do was to go back to the Manage Submissions page, click on the submission ID, and download the signed package (see **Figure 11, next page**).

That's all there is to it. Finished!

Just to make sure that Attestation Signing actually worked, we successfully installed the driver package on a newly installed RS1 system. It was nice to see that there was no pop-up asking "Do you trust this vendor?" – The installation procedure worked silently, just as it would if the driver had passed HLK testing.

We only had one question left: Will the Attestation Signed driver install on down-level version of Windows? To test this, we tried to install the driver on a newly installed Windows 8 system. And, predictably but unfortunately, this installation did not work as shown in **Figure 12 (next page)**.

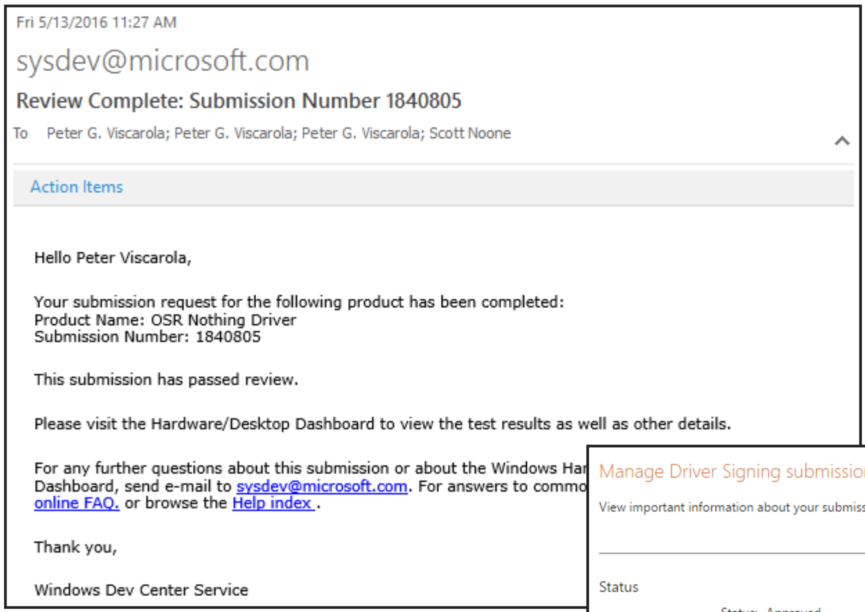
Wrap Up

So, in terms of signing, that's where things stand as of today. Life would be much easier if Microsoft

(CONTINUED ON PAGE 19)

Color Me Confused... (Cont.)

(CONTINUED FROM PAGE 18)



Fri 5/13/2016 11:27 AM

sysdev@microsoft.com

Review Complete: Submission Number 1840805

To Peter G. Viscarola; Peter G. Viscarola; Peter G. Viscarola; Scott Noone

Action Items

Hello Peter Viscarola,

Your submission request for the following product has been completed:
Product Name: OSR Nothing Driver
Submission Number: 1840805

This submission has passed review.

Please visit the Hardware/Desktop Dashboard to view the test results as well as other details.

For any further questions about this submission or about the Windows Har
Dashboard, send e-mail to sysdev@microsoft.com. For answers to commo
online FAQ, or browse the Help index .

Thank you,

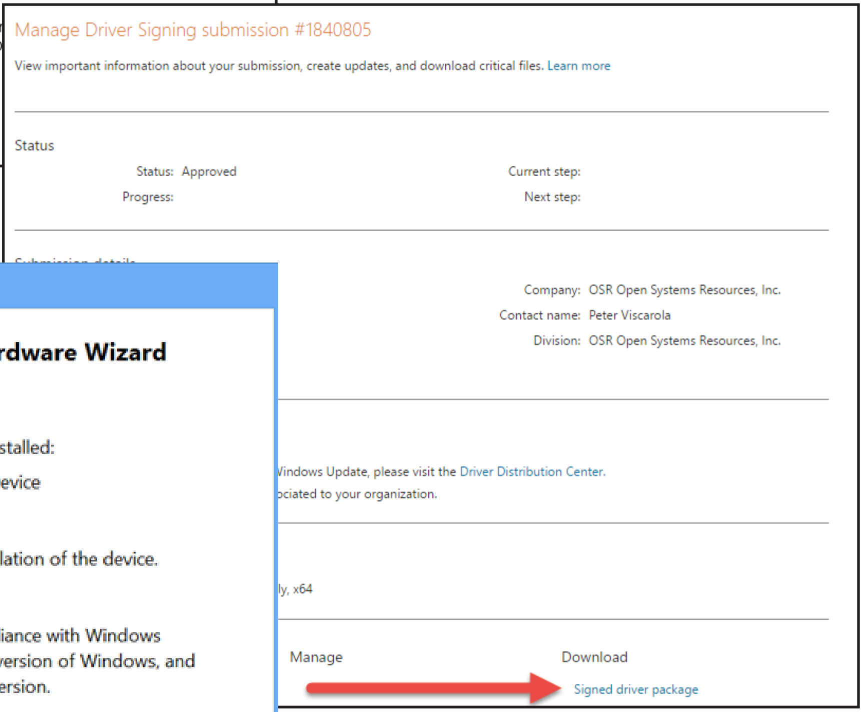
Windows Dev Center Service

Figure 10 — Fully Cooked in Less Than 30 Minutes

would just tell us what the signing policy is going to be. However, until that time, we get to spend our time testing out different theories.

We wish we were able to test a newly installed RS1 system for you… perhaps if an OEM or IHV would like to send us a system that supports Secure Boot, we can do that testing.

Follow us!



Manage Driver Signing submission #1840805

View important information about your submission, create updates, and download critical files. Learn more

Status

Status: Approved

Current step:

Progress:

Next step:

Company: OSR Open Systems Resources, Inc.

Contact name: Peter Viscarola

Division: OSR Open Systems Resources, Inc.

/indows Update, please visit the Driver Distribution Center.

ociated to your organization.

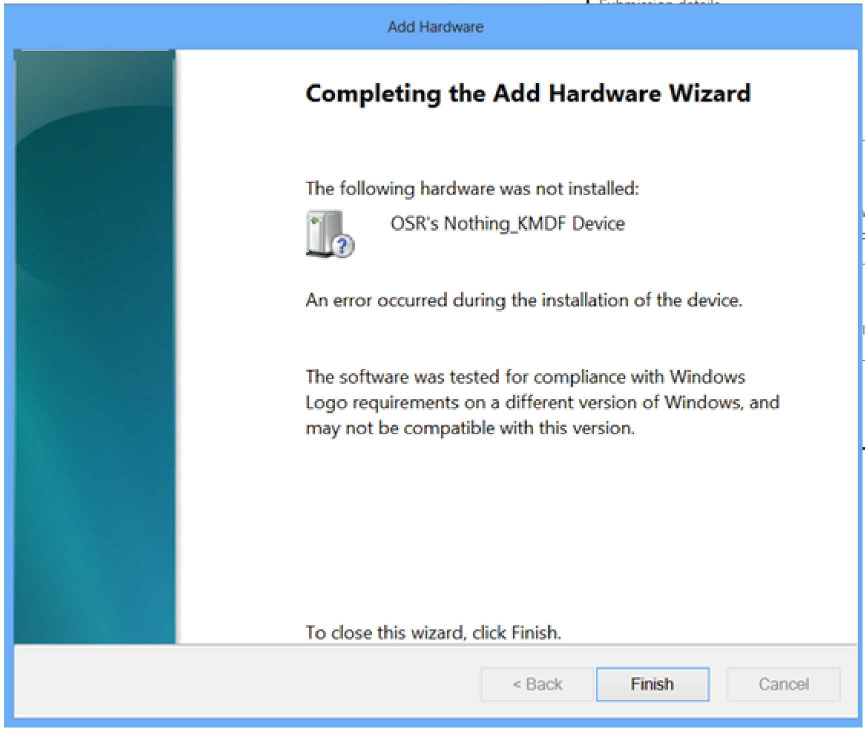
ly, x64

Manage

Download

Signed driver package

Figure 11 — Just Download the Package



Add Hardware

Completing the Add Hardware Wizard

The following hardware was not installed:

OSR's Nothing_KMDF Device

An error occurred during the installation of the device.

The software was tested for compliance with Windows Logo requirements on a different version of Windows, and may not be compatible with this version.

To close this wizard, click Finish.

Figure 12 — Attestation Signing is NO Help on Down-Level Operating Systems

A Few Rules... (Cont.)

(CONTINUED FROM PAGE 9)

the Request either synchronously or asynchronously with a callback using **WdfRequestSend**. So, if you want to learn one pattern for formatting and sending Requests, always call **WdfIoTargetFormatRequestForXxx** and **WdfRequestSend** for synchronous or asynchronous processing, and you'll always be right.

While you could stick with always using the same pattern, there **are** a few potentially useful optimizations that we should talk about that apply only to sending Requests to a Local I/O Target. The first of these is using _SEND_AND_FORGET. When you send a Request using **_SEND_AND_FORGET**, you're effectively telling the Framework to (a) send the Request to the Local I/O Target exactly as you received it and then (b) forget about that Request in terms of any further processing. The send operation is asynchronous. As soon as you call **WdfRequestSend**, you relinquish ownership of the Request, you do not get a callback when the Request is complete, and you're relieved of having to complete the Request in your driver.

_SEND_AND_FORGET is primarily useful for filter drivers that want to send along Requests that they receive from a Queue but do not need to process. For example, let's say you have a filter driver that's filtering IOCTLs. Your driver is interested in one specific IOCTL Control Code. If you get an IOCTL that doesn't have the Control Code you're interested in, you just want to pass it down the Device Stack to your Local I/O Target. You don't care if the driver below you completes the Request successfully. You don't ever want to see the Request again. You just want to pass the Request along, just as you received it.

```
//
// This control code is not interesting to us.  Just send the Request
// down to the next driver in the device stack. No formatting required!
//
WDF_REQUEST_SEND_OPTIONS_INIT(&sendOptions,
                              WDF_REQUEST_SEND_OPTION_SEND_AND_FORGET);

status = WdfRequestSend(Request, WdfDeviceGetIoTarget(myDevice), &sendOptions);

if (status == FALSE) {

    status = WdfRequestGetStatus (Request);

    WdfRequestComplete(Request, status);
}
```

FIGURE 2 — Sending to a Local I/O Target with _SEND_AND_FORGET

Because of the way it's typically used, **_SEND_AND_FORGET** allows the further optimization that you do not have to format the Request before sending it. It "just knows" to pass along the same Request parameters to your Local I/O Target that were passed into your driver. In fact, you **cannot** call **WdfIoTargetFormatRequestForXxxx** if you use **_SEND_AND_FORGET**. You can see an example of using the **_SEND_AND_FORGET** option in **Figure 2**.

_SEND_AND_FORGET is the lowest overhead way of using **WdfRequestSend** to send along a WDFREQUEST that you've received from a Queue to an underlying device and driver (your Local I/O Target) without modifying any of the Request parameters. Oh, by the way, you can only use **_SEND_AND_FORGET** with Requests that you got from a Queue. It won't work with Requests that you create in your driver using **WdfRequestCreate**.

```
//
// Pass along the same parameters that we received.
//
WdfRequestFormatRequestUsingCurrentType(Request);

//
// Completion callback… We need to know that the Request
// succeeded.
//
WdfRequestSetCompletionRoutine(Request,
                               FilterRequestCompletionRoutine,
                               WDF_NO_CONTEXT);

//
// Send the Request and don't wait for it to be completed
//
ret = WdfRequestSend(Request,
                     Target,
                     WDF_NO_SEND_OPTIONS);

if (ret == FALSE) {
    status = WdfRequestGetStatus (Request);
    WdfRequestComplete(Request, status);
}
```

FIGURE 3 — Formatting a Request "Using Current Type" for a Local I/O Target

A second optimization that you may wish to consider is the use of **WdfRequestFormatRequestUsing CurrentType**. You can use this function in many of the same cases where you might otherwise use **_SEND_AND_FORGET** but you want to specify either synchronous or asynchronous processing with a callback for your call to **WdfRequestSend**. You can see an example of the use of this function in **Figure 3**.

Note that it's also possible to call **WdfRequestFormatRequest UsingCurrentType** prior to sending a Request with **WdfRequestSend** specifying **_SEND_AND_FORGET**. While it may be architecturally valid as far as the Framework is concerned, doing this doesn't make a great deal of sense.

(CONTINUED ON PAGE 21)

A Few Rules... (Cont.)

(CONTINUED FROM PAGE 20)

Remember, **_SEND_AND_FORGET** automagically does the formatting, passing along the parameters sent to it. And it does it more efficiently than separately calling **WdfRequestFormatRequestUsing CurrentType**, too.

In Summary

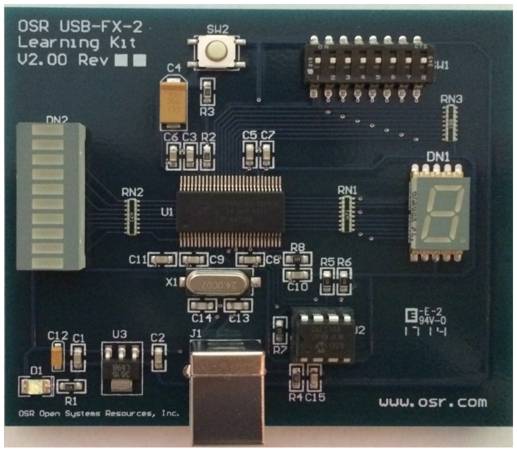
I hope you agree that if you view the rules in this way, it's quite easy to be sure you're using I/O Targets and **WdfRequestSend** the right way. In summary:

Remote I/O Targets

- Always format the Request being sent for the specific I/O Target using a function that starts with the characters **WdfIoTarget,** such as **WdfIoTargetFormatRequestForXxxx**. You may **not** use **WdfRequestFormatRequestUsingCurrentType**.
- Always call **WdfRequestSend** to send the Request using either synchronous or asynchronous processing. You may not use **_SEND_AND_FORGET**.
- If you use synchronous processing, be sure you've established a **WdfExecutionLevelPassive** constraint for your Device or Queue.

Local I/O Targets

- You may use any of the methods listed under Remote I/O Targets with Local I/O Targets. Methods for Remote I/O Targets will always work.
- As an optimization, if you want to send a Request you received from one of your WDF Queues to your Local I/O Target, and you do not want to change any of the parameters in the Request, **and** you do not need to see the Request after it is completed, you may choose to call **WdfRequestSend** using the **_SEND_AND_FORGET** option. This will send the Request to your Local I/O Target and effectively complete it from the viewpoint of your driver. If you do this, do not format the Request.
- Another possible optimization, if you want to pass along a Request that you received from one of your WDF Queues to your Local I/O Target **and** you do not want to change any of the parameters in the Request, but you **do** want to see the Request after it is completed, you may choose to format the Request using **WdfRequestFormatRequestUsingCurrentType**, and then call **WdfRequestSend** specifying either synchronous or asynchronous processing. Once again, if you use synchronous processing, be sure you've established a **WdfExecutionLevelPassive** constraint for your Device or Queue.



SPB Devices and Drivers... (Cont.)

(CONTINUED FROM PAGE 7)

channel, and one or more interrupts) as part of their **EvtDevicePrepareHardware** Event Processing Callbacks. In most respects, an SPB Controller driver is a pretty ordinary Windows driver. Because the driver for SPB Controller Devices are almost always written by the OEM/IHV and supplied as part of a system, there aren't many of these drivers written. As a result of all these factors, we won't discuss writing divers for SPB Controller Devices further in this article.

Drivers for SPB Client Devices resemble those for any device that's accessed via a protocol-based bus. An SPB Client Device driver opens a Remote I/O Target to its Controller Device, and interacts with its device by formatting WDFREQUESTs and sending them to the Controller Device.

In terms of hardware resources, a Client Device driver will always receive a "Connection ID" in the **EvtDevice PrepareHardware** Event Processing Callback. It might also receive one or more GPIO resources, which can be used to provide out of band data or interrupts from the Client Device to the driver.

The Connection ID is an opaque identifier that the Client Device driver uses to open a Remote I/O Target to the specific SPB Controller Device to which the Client Device is attached. The process of opening the Remote I/O Target to the correct Controller Device is accomplished with the assistance of another system component called the Resource Hub.

The code that a driver for a Client Device uses in its **EvtDevicePrepare Hardware** Event Processing Callback to create and open a Remote I/O Target given a Connection ID is shown in **Figure 4**.

```
case CmResourceTypeConnection: {

    WDF_IO_TARGET_OPEN_PARAMS  openParams;
    WDF_OBJECT_ATTRIBUTES targetAttributes;
    WDF_OBJECT_ATTRIBUTES_INIT(&targetAttributes);
    DECLARE_UNICODE_STRING_SIZE(resHubPath, RESOURCE_HUB_PATH_SIZE);

    //
    // Create the device path using the connection ID.
    //
    status = WdfIoTargetCreate(devContext->WdfDevice,
                               &targetAttributes,
                               &devContext->SpbControllerTarget);

    if (!NT_SUCCESS(status)) {

        // ...

    }

    //
    // Using the Connection ID, create the NAME pointing to the
    // Resource Hub.  The Resource Hub will resolve this open
    // by redirecting it to the appropriate Controller Driver
    // (the one to which our device is attached)
    //
    RESOURCE_HUB_CREATE_PATH_FROM_ID(&resHubPath,
        resourceTrans->u.Connection.IdLowPart,
        resourceTrans->u.Connection.IdHighPart);

    //
    // Open a Remote I/O Target to the SPB controller
    //
    WDF_IO_TARGET_OPEN_PARAMS_INIT_OPEN_BY_NAME(&openParams,
                                                 &resHubPath,
                                                 (GENERIC_READ | GENERIC_WRITE));

    status = WdfIoTargetOpen(devContext->SpbControllerTarget,
                             &openParams);

    if (!NT_SUCCESS(status)) {

        // ...
    }

    status = Bme280InitializeDevice(devContext);
    status = STATUS_SUCCESS;

    break;
}
```

Figure 4 — Code from EvtDevicePrepareHardware to open a Remote I/O Target to the Controller Device

In Figure 4, on receiving a Connection ID hardware resource you can see that the Client Device driver first creates an empty I/O Target object by calling **WdfIoTargetCreate**. Assuming the empty I/O Target is created successfully, the driver next builds a Resource Hub name with the macro RESOURCE_HUB_CREATE_PATH_FROM_ID passing in the received Connection ID (passed in the translated resources in **u.Connection.IdLowPart** and **u.Connection.IdHighPart**). This name is used during the I/O Target open

(CONTINUED ON PAGE 23)

SPB Devices and Drivers... (Cont.)

(CONTINUED FROM PAGE 22)

process to identify the Client Device to the Resource Hub when the I/O Target is opened. Finally, the driver calls **WdfIoTargetOpen** to open the Remote I/O Target.

More About Connecting SPB Client Devices to Controller Devices

Opening that Remote I/O Target to the Controller Device involves some pretty cool magic. The Connection ID that's built into the Remote I/O Target name uniquely identifies the Client Device to the Resource Hub. The Resource Hub, on receiving the open, checks the Client Device resources from ACPI and re-routes the I/O Target open operation to the correct Controller Device (**Figure 5**). This alleviates the Client Device driver from having to figure out which SPB controller instance its Client Device is connected to (and it is very common to have multiple SPB controllers in a system). In addition, when the Controller Device driver receives the open for the Remote I/O Target, it knows to which specific Client Device the open corresponds (via information from SPBCx). This allows the Controller Device driver to determine the bus address to use to communicate with the Client Device. Because the Controller Device driver has this information, the Client Device driver never needs to know (and in fact, generally cannot know) the address of its device on the bus. The bus address is completely handled by the Controller Device driver.

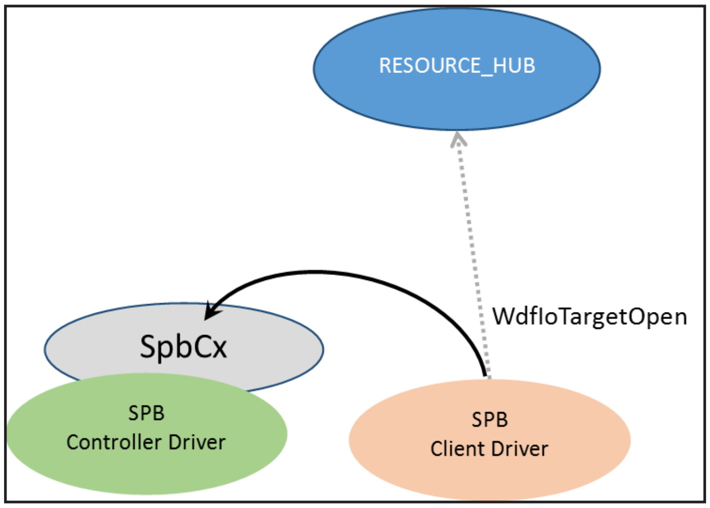


Figure 5 — Client Driver opens Remote I/O Target to Controller Driver via the Resource Hub

But there's another benefit to this Resource Hub and SPBCx integration mechanism. Because SPBCx provides a uniform interface for Client Device drivers to use to interact with the Controller Device driver regardless of whether the Client Device is connected via I^{2}C or SPI, making the connection via the Resource Hub and SPBCx also eliminates the Client Device driver from having to know the type of bus to which the Client Device is physically connected. Thus, if you're writing a driver for an IHV that makes a given sensor device, for example, and if that device can be connected via either I^{2}C or SPI (which is quite common), your driver doesn't have to change at all based on how the device is physically connected. How cool is **that**?

Reading and Writing Client Data

Once the Remote I/O Target is opened to the Controller Device, the driver for a Client Device can perform READ and WRITE operations on its device via the Controller Driver using **WdfRequestSend** to send ordinary read and write Requests. However, this isn't typical of how a driver interacts with an SPB Client Device. This is because the typical sequence of operations that the Client Device driver uses to read data from an SPB Client Device usually involves at least two operations. The most common pattern is:

(CONTINUED ON PAGE 24)

SPB Devices and Drivers... (Cont.)

(CONTINUED FROM PAGE 23)

1. The Client Device driver sends a write to the Remote I/O Target representing the Controller Device. This write is typically one byte in length and comprises a value indicating which register on the Client Device it wants to read;
2. The Client Device driver sends a read to the Controller Device's Remote I/O Target to read the data from the previously specified register. The number of bytes to be read is device specific and depends on the data that's being retrieved.

Pretty simple, right? The only trick that's involved is that the write and the read must generally take place in adjacent transactions on the SPB bus. That is, the Controller Device driver can't process the WRITE to the Client Device (to select the appropriate device register), then process a read or write to some other device on that same SPB bus, and then process the READ for the Client Device. That just won't work.

Again, SPBCx significantly simplifies things for the Client Device driver writer by providing an IOCTL design to perform the sequence described above. That IOCTL is **IOCTL_SPB_EXECUTE_SEQUENCE**. This IOCTL takes an **SPB_TRANSFER_LIST** in its Input Buffer. An **SPB_TRANSFER_LIST** contains a header and one or more **SPB_TRANSFER_LIST_ENTRIES**. Each **TRANSFER_LIST_ENTRY** contains a description of the direction of the transfer (that is, if the requested transfer is a write operation to the device or a read operation from the device), and a description of the data buffer to be used. The data buffer description can either be provided by a kernel virtual address and length in bytes, or an MDL. The number of **SPB_TRANSFER_LIST_ENTRIES** provided is indicated in the **SPB_TRANSFER_LIST_HEADER**. **Figure 6** illustrates an **SPB_TRANSFER_LIST**.

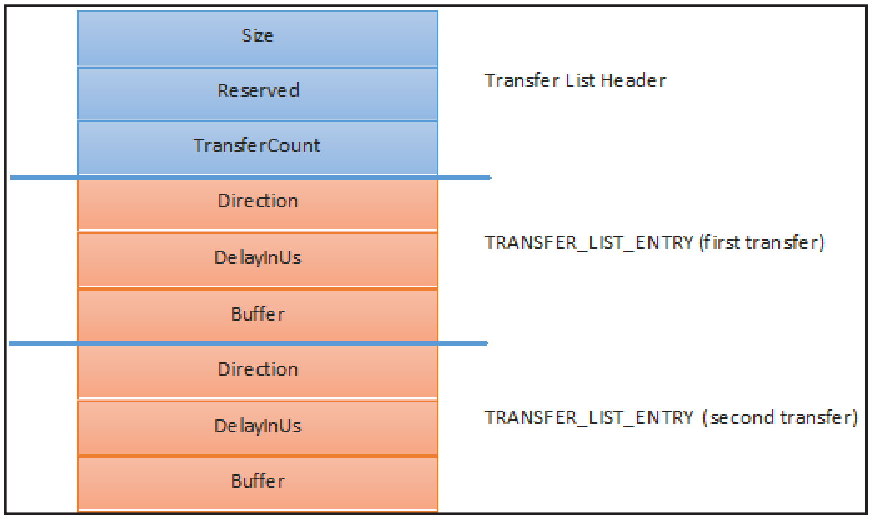


Figure 6 — SPB_TRANSFER_LIST with 2 SPB_TRANFER_LIST_ENTRYs

In Figure 6, you can see an **SPB_TRANFER_LIST** that describes a sequence of two transfers. The header for the Transfer List is shown in blue. Each transfer is described by a Transfer List Entry.

For example, to read four bytes starting at a specific register address on the Client Device, the driver would set the **TransferCount** field to 2, indicating that two **TRANSFER_LIST_ENTRY** structures would be used to represent the overall operation. It would set the first **TRANSFER_LIST_ENTRY** to represent the write of the register number to the Client Device. The Client Device driver would set the Direction field of the first **TRANFER_LIST_ENTRY** to **SpbTransferDirectionToDevice**, indicating a write operation to the device. The driver would put the Client Device register number from which it wanted to read into a buffer, and set the **Buffer** field of the first transfer list entry to point to that buffer, indicating a length of one byte. The second **TRANSFER_LIST_ENTRY** would then be set up to represent the read operation. The Direction in the second **TRANSFER_LIST_ENTRY** would be set to **SpbTransferDirectionFromDevice**, and the Buffer field of this **TRANSFER_LIST_ENTRY** would be set to point to a data buffer to hold the 4 data bytes read from the device.

You might also notice the **DelayInUs** field in each **TRANFER_LIST_ENTRY**. This field allows the driver to specify a minimum amount of time that should take place before a given transfer is initiated. One use for this delay is to allow a Client Device time to perform a specific operation that's been requested by one transfer in the sequence before, for example, starting a read for the results of that operation by a subsequent transfer in the sequence.

(CONTINUED ON PAGE 25)

SPB Devices and Drivers... (Cont.)

(CONTINUED FROM PAGE 24)

An example of a generic routine that will read a specified number of bytes from a given SPB device register is shown in the **OSRSpbReadRegisters** function in **Figure 7**.

```
_Use_decl_annotations_
NTSTATUS
OSRSpbReadRegisters(PBME280_DEVICE_CONTEXT DevContext,
                    UCHAR StartingRegister,
                    PVOID OutputBuffer,
                    ULONG OutLength)
{
    NTSTATUS status;
    WDF_MEMORY_DESCRIPTOR sequenceBufferDescriptor;
    ULONG_PTR bytesTransfered;

    // Allocate space for a 2 entry transfer list
    // for the Sequence of operations: WRITE followed by READ
    //
    SPB_TRANSFER_LIST_AND_ENTRIES(2)    tList;

    // Initialize the list
    //
    SPB_TRANSFER_LIST_INIT(&tList.List, 2);

    // Initialize the WRITE with the register number that we want to fetch
    // (this is just one byte)
    //
    tList.List.Transfers[0] = SPB_TRANSFER_LIST_ENTRY_INIT_SIMPLE(
        SpbTransferDirectionToDevice,
        0,                          // No delay
        &StartingRegister,
        sizeof(StartingRegister));

    // And initialize the READ with the place to store the returned value
    //
    tList.List.Transfers[1] = SPB_TRANSFER_LIST_ENTRY_INIT_SIMPLE(
        SpbTransferDirectionFromDevice,
        0,                     // No delay
        OutputBuffer,
        OutLength);

    // The send operation wants the buffer described with a MEMORY_DESCRIPTOR,
    // so that's what we build here.
    //
    WDF_MEMORY_DESCRIPTOR_INIT_BUFFER(&sequenceBufferDescriptor,
                                      &tList,
                                      sizeof(tList));

    // Send the sequence to the device: a 1 byte WRITE, followed by READ of the
    // amount of data specified.
    //
    status = WdfIoTargetSendIoctlSynchronously(DevContext->SpbControllerTarget,
                                               NULL,
                                               IOCTL_SPB_EXECUTE_SEQUENCE,
                                               &sequenceBufferDescriptor,
                                               NULL,
                                               NULL,
                                               &bytesTransfered);

    if (bytesTransfered != OutLength + 1) {

#if DBG
        DbgPrint("Bytes Transfered... expected 0x%0x, got 0x%0x",
                                                       (OutLength + 1),
                                                        bytesTransfered);
#endif
        status = STATUS_BAD_VALIDATION_CLASS;
    }

    return(status);
}
```

Figure 7 — A Generic Function to Perform Read Operations for an SPB Device Register.

In the example shown in Figure 7, the routine uses the macro **SPB_TRANFER_LIST_AND_ENTRIES** to allocate an **SPB_TRANSFER_LIST** with two **SPB_TRANSFER_LIST_ENTRY** structures. The Transfer List Header is then initialized using the **SPB_TRANSFER _LIST_INIT** function. The routine then initializes the two **SPB_TRANSFER_LIST _ENTRY** structures to describe each transfer: The first entry describes a one-byte write that contains the register number from which the read is to be performed. The second entry describes a read. Note that the routine uses the **SPB_TRANFER_LIST_ENTRY_INIT_ SIMPLE** macro to initialize each of these **SPB_TRANFSFER_LIST_ENTRY** structures, and describes the data buffer using a kernel virtual address and buffer length in bytes. The routine then builds a **MEMORY_DESCRIPTOR** to describe the buffer containing the **SPB_TRANSFER_ LIST** (because that's what the Send function that it uses wants). It then calls **WdfIoTargetSendIoctlSynchronously** to send the sequence to the Remote I/O Target that represents the Controller Device. Note that because it sends the sequence to the Remote I/O Target synchronously, this routine must be called at IRQL **PASSIVE_LEVEL**.

It's That Easy

With no complex configuration and the assistance provided by the SPBCx and the Resource Hub, writing a driver for an SPB device can be quite easy. As should be the case in WDF drivers, the

(CONTINUED ON PAGE 26)

SPB Devices and Drivers... (Cont.)

(CONTINUED FROM PAGE 25)

Framework and the available Class Extensions work together to make much of the interfacing details simple (see sidebar, *What are WDF Class Extensions?* below). This frees you up and allows you to spend your time determining how best to interact with your device and get your project done. That's not to say that everything about these devices is always simple, of course. For example, power management for SPB devices can sometimes be complex when these devices are integrated into system that support Modern Standby. But that's a topic for a whole other article.

In our WDF Core Concepts seminar, we spend time talking about the new buses that Windows supports, including SPB but also GPIO and async. If you'd like to learn more about writing drivers for these types of devices, we hope you'll join us.

Follow us!

What Are WDF Class Extensions?

Starting in Windows 8, the concept of WDF Class Extensions was introduced. Class Extensions provide a way to add support for a new class of device, such as SPB devices or NFC devices, into WDF without having to modify the underlying Framework itself. Class Extensions differ from other extended WDF support for (such as, for example, that provided for USB devices) because instead of the device class support being part of the Framework, it's supplied by an added DLL. This DLL, plus the Framework, plus the driver together form a complete entity.

Aside from the fact that they do not physically form part of the WDF Framework, what most quickly identifies a function as belonging to a Class Extension as opposed to the core WDF Framework is its name. The names of functions implemented by and structures defined by a Class Extension begin with an extension-specific prefix. For example, the functions provided by the SPB Class Extensions start with "Spb" (such as SpbRequestComplete, which are used by drivers for SPB Controller Devices, or the previously discussed SPB_TRANSFER_LIST structure). As a second example, functions provided by the NFC Class Extensions all start with "NfcCx" (such as the NfcCxDeviceInitialize function or the NFC_CX_SEQUENCE structure).

Other than the naming conventions and the fact that they do not physically form part of the WDF Framework, Class Extensions are pretty much indistinguishable from any other type of WDF support. Class Extensions can define unique Event Processing Callbacks that a driver can implement. They can also perform processing of standard WDF callbacks either in place of, or in addition to, those provided by a WDF driver.

Peter Pontificates... (Cont.)

(CONTINUED FROM PAGE 5)

the last time you were at a driver writer's conference that didn't include a heaping helping of Asian and/or Eastern European engineers? Ever wonder why? I'd say it's probably because these people actually learned something beyond the definition of "constructor" and "destructor" while in university. Talk to them sometime. Ask them what they learned in school. I bet before they could graduate they had to learn the difference between a North Bus and a bus heading north.

Before you go all "America First" on me, decide to put me in jail without the right to counsel, and accuse me of being an Al Qaeda sympathizer, please understand that I'm not saying that there are not some very good CS programs taught at some very good universities here in the United States. Several months ago on a plane, I actually sat next to a kid who was completing his junior year at CMU and actually knew what a spin lock was. Even more salutary was the fact that he knew why he'd want to use one. So, I suspect all is not lost. However, also note that this kid was sitting in first class. I wonder if this is significant.

We, here in the industry, have got to do something to try and stop the stupidizing of CS curricula in the States. That means you should do something about this, oh gentle and ever-lazy reader. If you're on an alumni committee, ask what's being taught in the CS department. Make your views known. If you teach at a University (even part-time), lobby to teach a real operating systems course to undergraduates. Push the department chair (who probably came up during the time when operating systems were still important and already harbors a closet resentment for the fact that assembler language isn't taught to freshmen) to talk to folks in the industry about our needs and revisit the graduation requirements.

If you're a recent CS grad, write to your former professors and tell them how well your education prepared you for the world of writing systems software. Explain to them that if they taught you the first thing about the fundamentals of computer science, you wouldn't have been so badly disadvantaged compared to your non-US trained colleagues. Copy and mail them this article. Heck, copy and send them this entire issue of The NT Insider just to see if they can understand anything more than this article and the letters section.

For the good of our industry, the current trend has to change. Either that or the rest of us will have to choose between getting lobotomized (so that we can make effective use of the Driver Wizard that will ship as part of Microsoft Office) and moving to Taiwan (where they'll still be writing drivers that talk to hardware). Me? I've already taken two semesters of Chinese, thank you.

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.

Follow us!

FILE ENCRYPTION SOLUTION FRAMEWORK

Develop Per-File Encryption Solutions Almost Exclusively in User Mode!

The OSR File Encryption Solution Framework (FESF) allows Clients to incorporate transparent, per-file encryption into their products—with the added benefit of development in user mode.

Beta releases of FESF are now available via a limited-access Early Adopter Program (EAP).

Learn more about FESF here, or contact the OSR Sales team at sales@osr.com.

OSR Seminar Schedule

Seminar	Dates	Location
Internals & Software Drivers	13-17 June	Dulles/Sterling, VA
WDF Drivers I: Core Concepts	25-29 July	**At OSR!** Amherst/Nashua, NH
Kernel Debugging & Crash Analysis	8-12 August	**At OSR!** Amherst/Nashua, NH
Internals & Software Drivers	12-16 September	Seattle, WA
WDF Drivers I: Core Concepts	3-7 October	**At OSR!** Amherst/Nashua, NH
WDF Drivers II: Advanced	11-14 October	**At OSR!** Amherst/Nashua, NH
Developing File Systems	7-10 November	Vancouver, BC

More Dates/Locations Available—See website for details

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...

"This was the third time I took an OSR course. For Windows kernel mode learning, I wouldn't go anywhere else. Even if I need to fly to the other side of the globe, I'd still go with OSR."

-Recent attendee of OSR WDF Core seminar

THE NT INSIDER - You Can Subscribe!

Just send a blank email to join-ntinsider@lists.osr.com — and you'll get an email whenever we release a new issue of The NT Insider.

Join OSRHINTS

Private Training

A private, on-site 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 on-site seminars are ideal.
- **Save money.** The quote you receive from OSR includes everything you need. There are never additional charges for materials, shipping or instructor travel.
- **Save more money.** Bringing OSR on-site to teach a seminar costs much less then sending several people to a public class. And you're not paying for your valuable developers to travel.
- **Save time.** Less time out of the office for developers is a good thing.
- **Save hassles.** If you don't have space or lab equipment available, no worries. An OSR seminar consultant can help make arrangements for you.