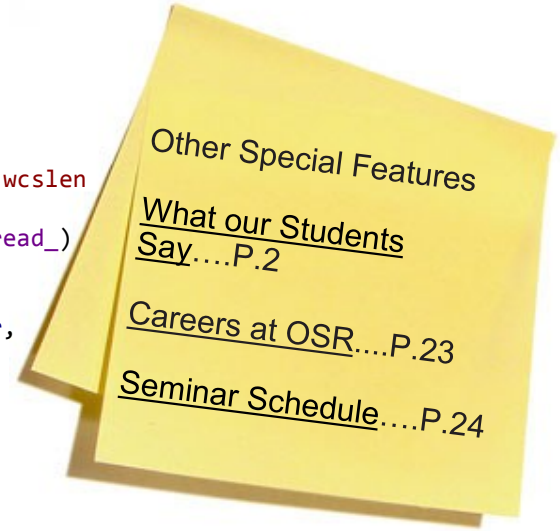
ISSUE VOLUME
02/20
MAY-JUNE
2014

The NT Insider

A publication of OSR Open Systems Resources, Inc.

```
// Real-Time Updates
_When_(PageNumber == 3, _Null_)
_When_(PageNumber == 3, _Notnull_)
_When_(PageNumber == 3, _Maybenull_)
PVOID
GetSocialWithOsr(
    _In_ ULONG PageNumber,
    _In_z_ _Notliteral_ _Null_terminated_ _Const_ PWCHAR Site
    );

// Peter Pontificates
_When_(PageNumber == 4, _Maybe_raises_SEH_exception_)
PVOID
WeWahHeeyaFirst(
    _In_ ULONG PageNumber
    );

// Using Bus Interfaces for Driver to Driver Communication
_Must_inspect_result_
_When_(PageNumber == 6, _Kernel_float_saved_)
PVOID
DontMissTheBus(
    _In_ ULONG PageNumber
    );

// The WDK Docs Improve Through Regular Releases
_Function_ignore_lock_checking_(_Global_cancel_spin_lock_)
_When_(PageNumber == 8, _Analysis_noreturn_)
PVOID
RightBeforeYourEyes(
    _In_range_(1, 14) ULONG PageNumber
    );

// Understanding EvtIoStop, Bugcheck 9F and Related SDV Errors
_Kernel_IoGetDmaAdapter_
_When_(PageNumber == 10,
       __drv_reportError("Caution: Reading this article may cause you to "
                         "actually understand why you need an EvtIoStop "
                         "(though it will do nothing to help you with "
                         "these SAL notations...)."))
VOID
NeededOrNiceToHave(
    _In_ ULONG PageNumber
    );

// Load or Unload
_When_(_Called_from_function_class_(FAST_IO_CHECK_IF_POSSIBLE),
                                    _Requires_no_locks_held_)
_When_(PageNumber == 12, _Kernel_clear_do_init_(__yes))
VOID
MakeUpYourMind(
    _In_ ULONG PageNumber
    );

// Drive Letter Alternatives
_At_(ArticleContents,
       _Writable_bytes_(_Inexpressible_((wcslen
(ArticleContents) + 1) * sizeof(WCHAR))))
_When_(PageNumber == 14, _No_competing_thread_)
VOID
DeathToDriveLetters(
    _In_ __drv_notPointer ULONG PageNumber,
    _Out_ PWCHAR ArticleContents
    );
```



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

Let's Give Them Something To Talk About

What our Students Say...about our WDF Seminar

"I was initially not very keen to attend but one of my manager's had taken the class and thought it was great so he wanted me and a colleague to take it. I found it much more interesting than I expected and feel like I learned a lot. Immediately after returning home I wrote a Windows driver for our current project.

- attendee of OSR's WDF Driver seminar (April 2014).

Scott was simply awesome. He did a very good job of making the class room training interesting and interactive.

- attendee of OSR's WDF Driver seminar (April 2014).

All or almost all modules and topics were right on and knowledge acquired is being used right away.

Very useful, worth every penny.

- attendee of OSR's WDF Driver seminar (April 2014).

Get Social with OSR
Real-Time Updates

Instead of waiting to provide you with information in the sporadic issues of *The NT Insider*, we at OSR have expanded our support of the community with more real-time updates. We don't necessarily cross-post everything to every social media site, so it would be best if you follow us on all the places you're active. We'll be offering special deals (such as discounts and events) for our followers. So, get connected!

The OSR Developer's Blog and RSS Feed – We've temporarily relocated our Dev Blog from the OSRONLINE community site to OSR.COM to prepare for an overhaul. You can read our Dev Blog at http://www.osr.com/developers-blog or subscribe to our RSS feed.

Twitter – We're increasingly using Twitter to mention things that we think are interesting and to and interact with you. For example, when *The NT Insider* is released our Twitter followers are the first to find out. Follow us @OSRDrivers.

LinkedIn – Another place the you can find out about what's up with us and ask questions. Check out our page at https://www.linkedin.com/company/osr and be sure to follow us.

Facebook – Yup, you can find us on Facebook too. Visit our page at https://www.facebook.com/OSRDrivers and follow us!

Real-Time Update
Here are a few of the things we've been talking about since the last issue of *The NT Insider* was published:

Peter discovers WDFSTRINGs – The fact that Peter discovered something he didn't know about WDF might be news enough, but the fact that he discovered the WDFSTRING abstraction and he doesn't think it sucks? Well, that's positively newsworthy. http://www.osr.com/2014/03/05/theres-a-wdfstring/

The !address kernel debugger extension works again for random kernel addresses. It's been broken since Vista, but it's back working: https://twitter.com/OSRDrivers/status/445910635902406656

Microsoft Released the WDK for Windows 8.1 Update… and it supports VS Express! If you haven't heard about **this** yet (you're not following us on Twitter or reading our blog and) you'll be excited to hear the news: Visual Studio Express now supports the WDK. Yay! Once again, all the tools necessary to develop Windows drivers are free. http://www.osr.com/2014/04/03/wdk-8-1update-wdk-now-supported-vs-express/

Dumping the crashing thread's stack at bugcheck time -- Know how to do that? We give you the command: https://twitter.com/OSRDrivers/status/456493578563620864

DMA Cache Coherency on ARM – You know how we're always beating on people to get them to not bypass the Windows DMA abstraction? And how they're always telling us, "it doesn't matter" because the whole world is like x86? Well, they need to think again: http://www.osr.com/2014/04/04/dma-cache-coherent-arm/

Microsoft Announces Sharks Cove Development Board – At Build, there was only one driver-specific session. But it was a good one ;-) In that session, Microsoft announced that (in partnership with Intel) they'd be releasing a Single Board Computer system and all the ancillary stuff needed to develop drivers for SPB-type peripherals. This is very cool, by itself. But what Peter *really* liked about this announcement wasn't just the hardware, it was the interesting change in Microsoft's approach to the driver development community that it heralded. Read what Peter has to say: http://www.osr.com/2014/04/07/msft-ready-engage-broader-driver-dev-community/

Look Through the WDK, and What Do You Find – You find some interesting details. Did you know that Windows 8.1 now has a dedicated interrupt stack? SNoone discovers several interesting details in header files starting with the prefix Kx. The details are all here: http://www.osr.com/2014/04/18/kx-headers-windows-8-1-wdk/

Now, ask yourself: Why haven't you been following us? If you had, you would have known about all this (and other stuff we didn't even bother to mention) in real-time.

Peter Pontificates:
We Wah Heeya First!

As you might have noticed, here at OSR we've been trying to hire an entry-level engineer or two for, oh, the last millennium or so. Give or take a few hundred years. And we've gotten lots of interesting letters/applications during that time.

The most intriguing ones have been from folks outside the US who would like to come work here. We've gotten a good number of resumes from folks who have a degree from a top-notch university, a good knowledge of kernel mode programming, and the ability to speak multiple foreign languages. Seriously, there was one guy who claimed to be fluent in English, Hindi, and Japanese. These folks all seem perfectly willing, even eager, to move to the Greater Boston area where OSR is located. Sadly, such stellar qualifications aren't sufficient to allow us to hire any of these folks. Being a small company, we just can't make it through the process to obtain the necessary visas, even for the most qualified candidate. Maybe we'll have to revisit that in the future, but for now, we've been looking for folks who are either working here in the States or in Canada or Mexico.

Aside from the aforementioned visa hopefuls, we've gotten a pile of resumes from guys (yes, they are specifically guys) with about 60 years' experience who used to work at DEC or Wang or Data General. They live in the Greater Boston area now, and some might even like to work here. Given that we already have a guy working here who used to work at DEC (that would be me, so don't start mouthing off) and a test and support engineering lead who used to work at DEC (that'd be Brenda), we've pretty much filled our quota for former DEC employees. But it was nice to hear from some of these folks just the same.

Aside from these two groups of folks we've gotten… oh… really close to zero applicants.

Now, I'm not saying we've gotten *exactly* zero interest. Just *close* to zero interest. There have been a few nibbles. I met and talked with one really exceptional woman who was seriously intrigued by the kind of work we do, had the best attitude I could have hoped for, and was even darn well qualified. I met and spoke with her on one of my many trips to the West Coast. After our talk, I was ready to immediately fly her back to OSR for a real interview. Then it hit the fan:

She: *Sounds so cool! So where are you located exactly?*

Me: *Well, we're in a little town called Amherst, New Hampshire… about an hour from Boston. It's really lovely, and we're near the mountains and the ocean and…*

She: *Hmmm… My fiancé will be finishing school soon. Are there lots of computer companies in your area?*

Me: *Well, ah, there's EMC, and Citrix, and ah… Stratus. And, I dunno…Oracle. Does Fidelity count?*

She: *Hmmm… I'll talk with my fiancé and see what he says, and I'll let you know.*

Queue the crickets. That was the last I heard from her. Too bad. For us, of course. I'm sure she got an awesome job with founder stock at some company in Silicon Valley.

Figure 1— Downtown Amherst, NH

(CONTINUED ON PAGE 5)

Peter Pontificates... (Cont.)

(CONTINUED FROM PAGE 4)

I asked a buddy of mine out at Microsoft why *he* thought we weren't getting more applicants:

Me: *Do we stink or something? Why do you think we're not seeing more people applying?*

Buddy: *Duh!*

Me: *Duh, WHAT!*

Buddy: *Duh…Look at where you're located.*

Me: *Well, I admit, we're not in the most gorgeous building in the universe, but…*

Buddy: *No, fucktard, not your building (rolls eyes). You're in New Hampshire. Almost nobody even knows where New Hampshire is. (thinks for a minute) They probably know that it's cold. And it's not in Silicon Valley.*

Me: *What do you mean they don't know where New Hampshire is?! We're just North of Boston. Harvard. MIT. These are actual schools that actual people have actually heard of.*

Buddy: *They're in Cambridge. And it doesn't matter. Nobody wants to work in New Hampshire. There is no high tech in New Hampshire. Or Massachusetts. Or in any of those other tiny states over there. Suppose they come work for you, discover what an asshole you are, and want to change jobs? How would they interview? Fly across the fucking country?*

Me: *Have these people never heard of Skype? If not, they should try it. I've got a friend from here on the East Coast who got a sweet job with stock at one of the hottest VC funded startups in all of tech. He did all his interviewing via Skype. He never saw the place in person until his first day of work.*

Buddy: *He was from the East Coast, you said?*

Me: *Yup!*

Buddy: *And he got a great job at a tech startup? With tons of stock?*

Me: *Yup! Yup! (sits back in a satisfied way)*

Buddy: *Where was this job located?*

Me: *Ah, Mountain View, I think.*

Buddy: *See?! You're gonna need to move your company to someplace civilized. Have you considered Mountain View?*

We're not moving OSR anytime soon. The Boston area is already very civilized. Where else will you find people who talk with an accent that's as cool as the one here in Boston? Silicon Valley might be the flavor of the month (OK, flavor of the past three decades… whatever) but, as my friend Tommy from Quinzee would say, "That is nawt faayah! We wah heeyah first! No one denies this!" Before there was a Silicon Valley, the Boston area had Route 128. Heck, Route 128 is America's Technology Highway! Just take a look at **Figure 2**, a page from Wikipedia about the Famous Route 128, with a few notes I made.

(CONTINUED ON PAGE 23)

The high-tech region [edit]

In 1955, *Business Week* ran an article titled "New England Highway Upsets Old Way of Life" and referred to Route 128 as "the Magic Semicircle". By 1958, it needed to be widened from six to eight lanes, and business growth continued, often driven by technology out of Harvard University and MIT.[3] In 1957, there were 99 companies employing 17,000 workers along 128; in 1965, 574; in 1973, 1,212. In the 1980s, the area was often compared to California's Silicon valley,[4][5] and the positive effects of this growth on the Massachusetts economy were dubbed the "Massachusetts Miracle".

Major companies with significant locations in the broader Route 128 area included:

- ~~Digital Equipment Corporation~~ dead
- ~~Data General~~ dead
- BBN Technologies shadow of former self
- Thermo Electron and Fisher Scientific, later merged as Thermo Fisher Scientific
- Analog Devices
- ~~Computervision~~ dead
- ~~Microsoft~~ [citation needed] never here…Wikipedia sucks
- ~~GTE~~ dead
- ~~Honeywell Information Systems~~ effectively dead
- MITRE vastly reduced
- ~~Polaroid~~ shake it like a…
- ~~Sun Microsystems~~ both experiencing the afterlife under Ellison
- ~~BEA Systems~~
- Turbine, Inc. actually, a cool gaming company
- EMC Corporation
- Autodesk Autodesk?
- Raytheon military
- Wang Laboratories
- Apollo Computer all dead
- Prime Computer
- Cullinet

Figure 2— From Wikipedia: America's Technology Highway!

Don't Miss the Bus
Using Bus Interfaces for Driver-to-Driver Communication

While putting together some new material for our Advanced WDF Driver Development seminar, we began taking yet another look at one of our favorite topics around here: driver to driver communication. As with every other topic related to Windows driver development, the seemingly innocuous task of calling from one driver into another quickly devolves into an endless series of options, features, and arcane WDM trivia.

However, a shining light came out of these discussions: the venerable Bus Interface architecture is still alive and kicking in Windows. Not only does it provide a clean, well defined method of driver to driver communication, but KMDF provides helper routines to make producing or consuming one of these interfaces a breeze.

What's a Bus Interface?

As the name would imply, a Bus Interface is a standard way for a Bus driver to provide a procedure call interface to its children. A Bus driver can have multiple Bus Interfaces, each of which is identified via a GUID. Consumers of the Interface query for the interface using PnP IRPs and are returned a data structure defined by the Bus, which can include any data or functions that the Bus driver wants to share (**Figure 1**).

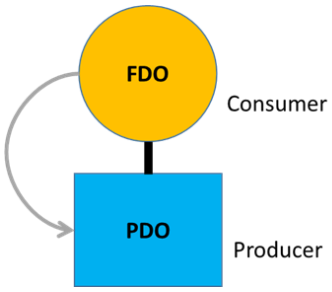


Figure 1

When a Bus driver provides a Bus Interface for its children, the expectation is that only drivers *within the PDO's device stack* will consume the Interface. One reason for this is that it eliminates any possible race conditions in the teardown case. Stacks are destroyed from the top down, thus the PDO is always the last device to be deleted. If the Bus Interface is only consumed by drivers higher in the stack, then we don't need to worry about anyone trying to use the Bus Interface after the PDO has been deleted.

Bus Interfaces Aren't Just for Buses

Clearly Bus Interfaces are generically useful outside of Bus drivers, driver to driver communication is common and not *all* driver to driver communication scenarios simply involve interacting with your own PDO. Thankfully, the Bus driver support in the operating system is generic, thus it is possible for a filter or FDO to process the appropriate PnP IRPs and publish their own Bus Interface. This means we could have a consumer using a Bus Interface produced by a different device stack (**Figure 2**).

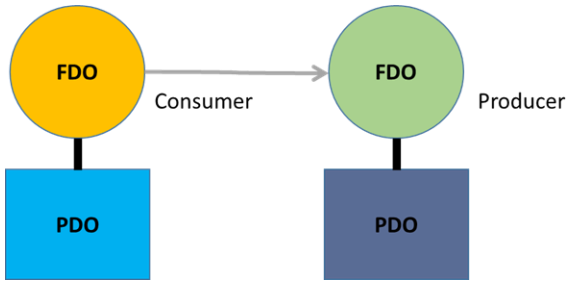


Figure 2

It also means that we could consume a Bus Interface of a producer *above our device* in the stack (**Figure 3**).

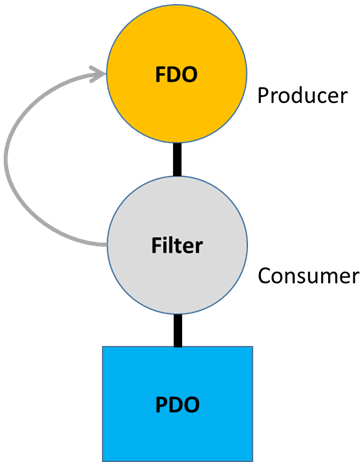


Figure 3

For these reasons, the WDK documentation has taken to using the term *Driver Defined Interface* as opposed to Bus Interface, so expect to see documentation referring to this feature using both terms.

Beware that there is a hidden complexity in using a Bus Interface produced by a device other than your PDO: teardown. Either the consumer or the producer of the interface *must* guarantee that the producer will not be removed while the consumer is still using the Bus Interface. Failure to do so will result in the consumer calling into an unloaded driver, which will result in an immediate bugcheck.

Providing a Bus Interface in KMDF

As with so many other things, KMDF makes providing a Bus Interface for our function, filter, or physical device object a

(CONTINUED ON PAGE 7)

Using Bus Interfaces... (Cont.)

(CONTINUED FROM PAGE 6)

breeze. Thankfully, the Framework makes no distinction in its API for which type of device the producer is. Instead, the WDFDEVICE object provides the **WdfDeviceAddQueryInterface** method to add a Bus Interface to an existing WDFDEVICE (**Figure 4**)

```
_Must_inspect_result_
_IRQL_requires_max_(PASSIVE_LEVEL)
NTSTATUS
WdfDeviceAddQueryInterface(
    _In_ WDFDEVICE Device,
    _In_ PWDF_QUERY_INTERFACE_CONFIG InterfaceConfig
    );
```

Figure 4

The driver-provided **WDF_QUERY_INTERFACE_CONFIG** structure defines both the GUID and the interface structure to the Framework, which are both specified when the structure is initialized with **WDF_QUERY_INTERFACE_CONFIG_INIT** (**Figure 5**).

```
VOID
FORCEINLINE
WDF_QUERY_INTERFACE_CONFIG_INIT(
    _Out_ PWDF_QUERY_INTERFACE_CONFIG
                        InterfaceConfig,
    _In_opt_ PINTERFACE Interface,
    _In_ CONST GUID* InterfaceType,
    _In_opt_ PFN_WDF_DEVICE_PROCESS_QUERY_INTERFACE_REQUEST
                    EvtDeviceProcessQueryInterfaceRequest
    );
```

Figure 5

The Framework then responds to the PnP IRPs sent by the consumers, returning the driver defined interface structure. The driver may optionally specify an *EvtDeviceProcessQuery Interface* event processing callback to be notified any time a consumer requests the interface from the producer.

Contrary to what you might expect, the **WDF_QUERY_INTERFACE_CONFIG_INIT** macro does not simply take a PVOID and length for the interface structure. Instead, it takes a pointer to an O/S defined **INTERFACE** structure. This **INTERFACE** structure is a required header for any Bus Interface provided by a producer. The **INTERFACE** structure definition can be found in **Figure 6**.

```
typedef struct _INTERFACE {
    USHORT Size;
    USHORT Version;
    PVOID Context;
    PINTERFACE_REFERENCE InterfaceReference;
    PINTERFACE_DEREFERENCE InterfaceDereference;
    // interface specific entries go here
} INTERFACE, *PINTERFACE;
```

Figure 6

Per the comment in the structure definition, this common header is followed by the driver-defined portion of the structure. The **INTERFACE** structure has the following required members:

(CONTINUED ON PAGE 17)

Right Before Your Eyes
The WDK Docs Improve Through Regular Releases

Some stuff that happens in the driver world is a big deal, like the WDK becoming integrated with Visual Studio. Or, SDV finally becoming so useful all you want from it is more checking. Or, UMDF and KMDF sharing the same syntax. These are all Big Events that matter to just about every driver writer on the planet. They thus get big news, and lots of people blog, write articles, Tweet, whine, praise, and make pithy comments like "what took you so long." But, one way or the other, most people in the driver development world hear about these big changes.

What people probably don't realize is that there are other changes that happen that are just as important to the community, but that almost nobody talks or hears about. One of these is the steady progression of the WDK documentation over the days, weeks, months, and years.

When the Docs ~~Sucked~~ Weren't ~~Any Good~~ So Great

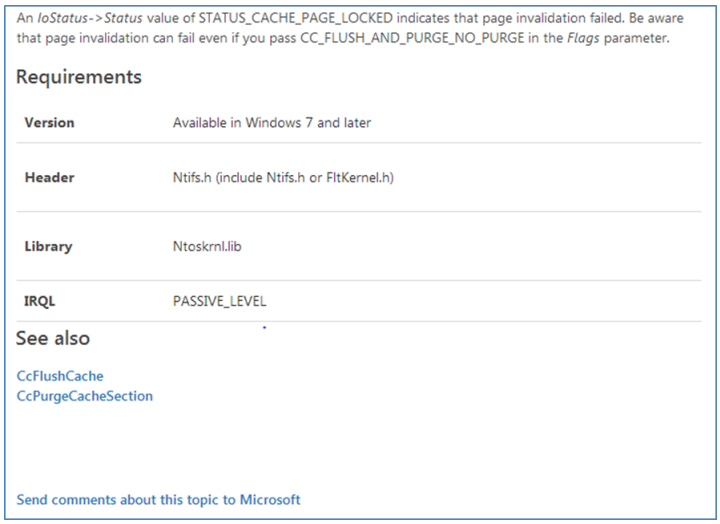
If you've been involved in Windows driver development for a while, you might remember when the entire set of driver development documentation fit into two printed volumes totaling less than 500 pages. There were many, many, basic driver development interfaces that weren't documented at all. The ones that **were** documented varied tremendously in their clarity and level of detail. Heck, for quite a while, I had a standard "call and response" I used in the Windows driver development classes I taught. As I explained a certain topic or function, I would say "But that's…" and then I'd gesture to the class. The class would then complete my phrase in unison: "NOT DOCUMENTED!" It was great fun, and it kept (almost) everyone awake during my otherwise scintillating lectures.

That was quite a while ago. But even in more recent times, it was common that features introduced in a new operating system version weren't necessarily documented at the same time the OS including those features was released. And for **ages** it was standard practice that the WDK docs were released only when the WDK was released -- once per OS or service pack release. Any changes, additions, or clarifications to the docs had to wait for the next release cycle. This was true even after the WDK docs were officially available online on the Microsoft site.

Now? A World of Difference

But that was then, and this is now. I came today to praise the WDK docs and the team that creates them, not bury them. I want to call attention to some very cool things that are happening right before your eyes. Things that you might not have noticed unless you were looking carefully.

For the past few years, the WDK docs have been undergoing what I can only refer to as a "continuous improvement" program. You know that link that lets you send feedback at the bottom of each doc page (see **Figure 1**). They actually **read** that feedback. They'll file **bugs** based on it. How do I know this? I've gotten replies… and bug resolutions… and actual documentation fixes… based on comments I've sent using those links. Really!


An *IoStatus->Status* value of STATUS_CACHE_PAGE_LOCKED indicates that page invalidation failed. Be aware that page invalidation can fail even if you pass CC_FLUSH_AND_PURGE_NO_PURGE in the *Flags* parameter.

Requirements

Version	Available in Windows 7 and later
Header	Ntifs.h (include Ntifs.h or FltKernel.h)
Library	Ntoskrnl.lib
IRQL	PASSIVE_LEVEL

See also

CcFlushCache
CcPurgeCacheSection

Send comments about this topic to Microsoft

Figure 1— Yes, The Feedback Link Does Work

But that's not the best part. Sure, fixing reported bugs is cool. But the major change you might not notice is that instead of being mostly reactive – fixing content problems when somebody reports a problem – the doc team has now gotten to the point where they're being very **pro**active. They're looking at doc content, without anybody specifically complaining about it, and seeing how it can be improved and made easier to find.

(CONTINUED ON PAGE 9)

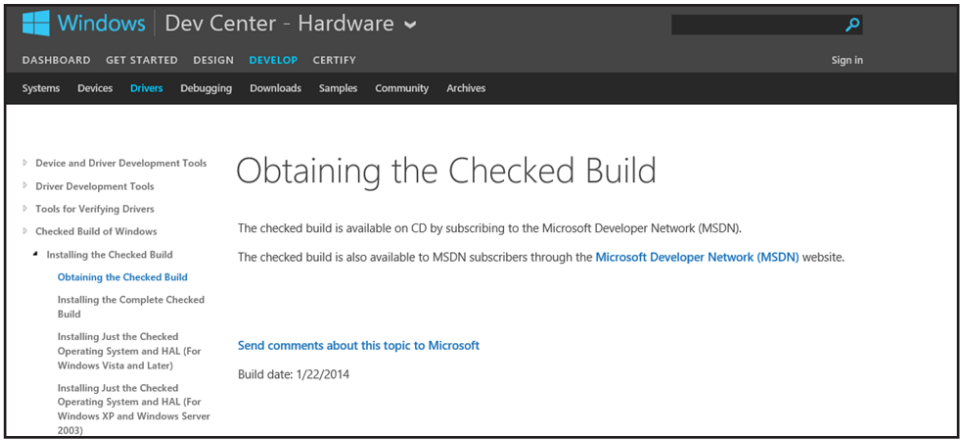


Figure 2— The Old Page. Booooo!

The WDK Docs Improve... (Cont.)

(CONTINUED FROM PAGE 8)

Need an example? Here's one: Let's look at the documentation about a nice, simple, topic. The Checked Build of Windows. **Figure 2** (previous page) shows the WDK doc page as it appeared at the beginning of this year (2014).

Not that informative, is it. But the information **is** there. Sort of. Some of it. But, looking at this page and having just finished helping with a major update to the OSR.COM corporate web site, I can't help but think of this page in SEO ("Search Engine Optimization") terms. How many people who are looking for the checked build of Windows Google "Obtaining the checked build of Windows"? Hey… the term "Windows" *isn't even on the page*. Good luck finding this if you're doing a web search.

And now we turn to **Figure 3**, the analogous page from the WDK as of 1 May 2014. Not only is the content useful and up to date, there's actually a chance someone can **find** this page with a search. In fact, when I Google "How do I get a checked build of Windows" this page turns up in the first three results. The other two are other pages in this same group of topics.

It seems that the WDK documentation is now being updated regularly. I don't know exactly **how** frequently it's updated, but it seems to me to be pretty frequently. And this leads to an interesting issue: For many of us, after installing Visual Studio and the WDK, one of the next things we do is download the offline WDK docs. You know, in case we want to write code when you're offline (like when you're on a United Airlines flight from Boston to San Francisco, where the plane was made in like 1950, with little TV sets that drop down from the ceiling every 10 rows for you to watch the movie, and no room to move, and there's no in-flight WiFi, and… oh, wait. Never mind). While the offline docs are still useful, the advantage to using the online docs is that you get the "latest and greatest" version of the docs when you access them. And now that the docs are updated frequently, that can be a significant advantage.

More Goodness

There's more to notice and to like about how the WDK docs have progressed over the last few years. Information on features published contemporaneously with the OS that introduces those features (did you see that nice clear documentation about SPBs appeared in the RTM Windows 8 doc set?), more "how to" and "step by step" guidance for new driver writers, and more architecture, context, clarity, and usefulness in the docs for everyone, old hands included.

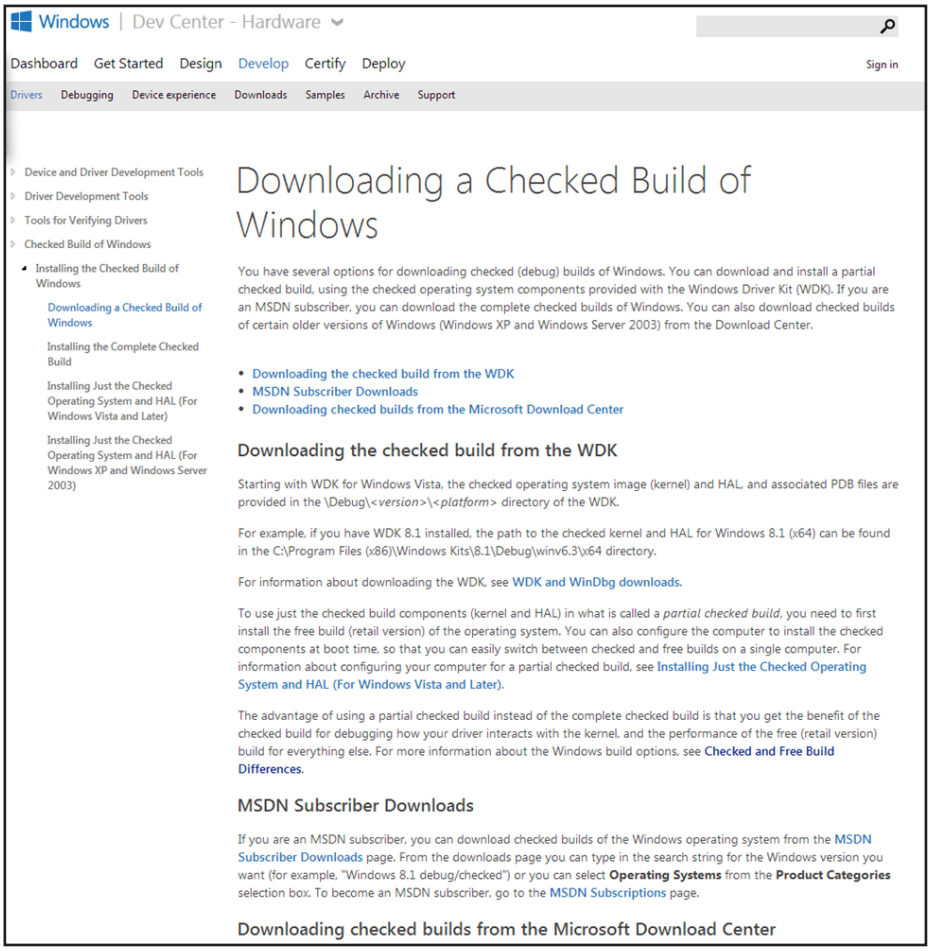


Figure 3— A Bountiful New Page (partial page shown). Yay!!

Here's a good example of changes in this last category: Clarity and usefulness. The description in the WDK of the IRP's **Tail.Overlay.Thread** field from not that long ago. Read it, and tell me if you understand what it means:

Tail.Overlay.Thread

Is a pointer to the caller's thread control block. Higher-level drivers that allocate IRPs for lower-level removable-media drivers must set this field in the IRPs they allocate. Otherwise, the FSD cannot determine which thread to notify if the underlying device driver indicates that the media requires verification.

Got that? No!? Wonder WTF it's talking about? Yeah, me too. It's so narrow it's just barely correct. Seriously. That **was** the description of this field. Do you wonder why we got questions about this field on NTDEV all the time? Compare that to the definition I just got online while I was writing this article (May 2014):

(CONTINUED ON PAGE 21)

Needed or Nice-to-Have?
Understanding EvtIoStop, Bugcheck 9F, and Related SDV Errors

Barely a week passes without somebody posting on the NTDEV list about power state transition failures related to WDF Queues. "I can't stop my Queue" or "When I attempt to set my system into Standby, I sporadically get a bugcheck 0x9F", are the most common issues I hear. I also hear about people getting SDV errors for drivers that passed in Win7 and have been otherwise working for ages. Or, I get an email from a former student asking about the (excessively long yet surprisingly) cryptic comment and its associated **_analysis_assume** shown in **Figure 1**, which appears close to twenty times in the driver examples in recent WDKs.

```
//
// By default, Static Driver Verifier (SDV) displays a warning if it
// doesn't find the EvtIoStop callback on a power-managed queue.
// The 'assume' below causes SDV to suppress this warning. If the driver
// has not explicitly set PowerManaged to WdfFalse, the framework creates
// power-managed queues when the device is not a filter driver.  Normally
// the EvtIoStop is required for power-managed queues, but for this driver
// it is not needed b/c the driver doesn't hold on to the requests for
// long time or forward them to other drivers.
// If the EvtIoStop callback is not implemented, the framework waits for
// all driver-owned requests to be done before moving in the Dx/sleep
// states or before removing the device, which is the correct behavior
// for this type of driver. If the requests were taking an indeterminate
// amount of time to complete, or if the driver forwarded the requests
// to a lower driver/another stack, the queue should have an
// EvtIoStop/EvtIoResume.
//
__analysis_assume(ioQueueConfig.EvtIoStop != 0);
status = WdfIoQueueCreate( device,
                           &ioQueueConfig,
                           WDF_NO_OBJECT_ATTRIBUTES,
                           &hQueue );
```

Figure 1—What's with this _analysis_assume?

D-State Changes with Requests in Progress

All these issues are related to the simple fact that when the system wants to transition a device out of D0 (Working) State to a lower-power D-State, any "active" I/O requests on that device must be "accounted for" by the driver.

The fact that outstanding Requests need to be considered when your device is powering off is a pretty basic concept. When your device is being powered-off, if you have read or write Requests in progress you're going to have to do **something** with those Requests. You need to abort them (returning an error to the user), stop them and restart them again when you device powers back up, or… something. You need to do this because one thing you can count on is the fact that your device won't be completing that Request while it's powered off.

Back in the caveman days of WDM, it was entirely up to you to figure this out and manage the process. Drivers "losing" I/O operations across power state transitions wasn't exactly unheard of. Fortunately, we no longer have to deal with WDM any more than we have to find and club our own food. The WDF Framework once again has come to rescue us from the Stone Age.

So, what happens in WDF? In WDF by default any WDF Queues that you create are "power managed." When a WDF Queue is power managed, it means that the state of the Queue follows the power state of its associated device. Specifically, whenever the device is powered-on (in D0 State) the Queue will present Requests to the driver based on the Queue's Dispatch Type. For convenience, we'll say the Queue is in the **Started** state. When the device attempts to transition from powered on (D0) to a lower powered state (Dx), all power managed Queues for the device will no longer present Requests to the driver. Once again, for convenience, we'll refer to this as the Queue being in the **Stopped** state.

When the system wants to transition your device from D0 to Dx, it tells the WDF Framework. The Framework (does a bunch of stuff and then) attempts to transition each of your device's power managed Queues from **Started** to **Stopped** state. And this is where the misunderstanding frequently begins. One of the key points people often miss is that *the Framework will delay the device's D0 to Dx power state transition* until all of the device's power managed Queues have successfully entered the **Stopped** state. If that delay lasts too long, the system bugchecks with a 0x9F crash code.

Stopping WDF Queues

Simple, right? Your device needs to go to Dx, but before it can do so, the Framework moves all its power managed Queues to the **Stopped** state. So, you may ask, what needs to happen before a Queue can enter **Stopped**?

Before WDF can successfully transition a power managed Queue to **Stopped**, all Requests that have been presented to your driver from that Queue *and are still active* need to be accounted for. This is how WDF helps you ensure that you don't lose any I/O requests across a power-state transition. The Framework considers a Request to be "active" until your driver has done one of the following things with it:

- Completed it by calling **WdfRequestComplete** (or some variant of that function)
- Sent it to a Remote I/O Target using Send And Forget
- Forwarded it to another Queue

If that list looks familiar, it's because it's the exact same list of items that will trigger the release of a new Request from a Queue with Sequential Dispatching. Consistency is good, right? Given the above, if your driver's been presented with a read Request, and that Request is currently in progress on your device, that Request will need to be accounted for by your driver before the Queue that presented that Request can be **Stopped**. Likewise, if you have a Request that you've sent asynchronously with **WdfRequestSend**, that Request will need to be accounted for by your driver before its Queue can be **Stopped**.

(CONTINUED ON PAGE 11)

Understanding EvtIoStop… (Cont.)

(CONTINUED FROM PAGE 10)

Accounting For In-Progress Requests

How does a driver properly "account for" Requests that are active when a device power state transition is pending? Well, the traditional way has been to not worry much about it. That is, the traditional way of accounting for pending Requests during a power-state transition is to simply wait for the Requests to complete. When all the Requests that are active from the Queue are completed (by the driver calling **WdfRequestComplete** or similar), the Queue can be stopped and then the Framework will allow the system to finish transitioning the device to Dx.

Depending on your driver and device architecture, there can be some significant disadvantages to this traditional approach. For one, if your device takes **any** significant amount of time to complete its in-progress Requests, you could slow down the entire system's transition to a lower power state. Not to mention the fact that no user wants to sit watching their tablet/ultrabook/notebook/PC while it takes its time suspending. Another disadvantage to the traditional "do nothing" approach is that there are lots of Requests that can remain in progress for a long or indefinite amount of time. Consider, for example, a read operation from a serial data device. The read won't complete until data has arrived. Until that time, the Request sits and waits – in progress the entire time.

And this, my friends, is where the **EvtIoStop** Event Processing Callback comes in. When it attempts to transition a WDF Queue to **Stopped** state, the Framework calls **EvtIoStop** for each active Request that's been presented to the driver from a power managed Queue. Within its **EvtIoStop** function, the driver "accounts for" the Request by doing one of the following things:

- Making the Request no longer active, by:
  - Completing the Request with whatever status makes sense (can be success or failure).
  - Successfully sending the Request to an I/O Target using the Send And Forget option.
  - Successfully forwarding the Request to another WDF Queue belonging to the device.

- Asking the Framework to put the Request back on the Queue from which it came. The driver can do this by calling **WdfRequestStopAcknowledge** with the **Requeue** set to **TRUE**.

- Telling the Framework that it intends to keep the Request in progress by calling **WdfRequestStopAcknowledge** with the **Requeue** parameter set to **FALSE**.

- Successfully canceling the Request by calling **WdfRequestCancelSentRequest** if the Request has been previous sent to an I/O Target without using the Send And Forget option.

Calling you at **EvtIoStop** is the Framework's way of reminding you of the Requests that you have in-progress from a given Queue, before that Queue can become **Stopped** and the device can transition to Dx. Within your **EvtIoStop** routine you handle (or tell the Framework to handle) each pending Request. In this way, each active Request is "accounted for" in a timely way, no Requests are "lost" across a power-down event, and the device's transition to a lower-powered D-State is not delayed.

(CONTINUED ON PAGE 22)

Make Up Your Mind
Load or Unload

An interesting crash we have seen relate to a scenario that involves unloading the driver at the same time it is being loaded. In this article, we'll analyze what we saw, how we reached the conclusion we reached and the remedial steps we used to attempt to mitigate against this particular problem.

The Crash

We were recently given a crash dump from a system that had been under test with a file system filter driver that performs isolation – that is, it controls the cache and uses shadow file objects to distinguish between the resources that **it** controls and the resources that belong to the underlying file system (typically NTFS).

Analyzing the crash with WinDBG, we found two interesting threads. Here's the first:

```
     THREAD fffffa80018b8b50  Cid 0004.0044  Teb: 0000000000000000
Win32Thread: 0000000000000000 RUNNING on processor 2
        Not impersonating
        DeviceMap                 fffff8a0000060f0
        Owning Process            fffffa8001844840       Image:         System
        Attached Process          N/A            Image:         N/A
        Wait Start TickCount      264493         Ticks: 1 (0:00:00:00.015)
        Context Switch Count      60091          IdealProcessor: 2
        UserTime                  00:00:00.000
        KernelTime                00:00:01.263
        Win32 Start Address nt!ExpWorkerThread (0xfffff80002ad8530)
        Stack Init fffff88003195db0 Current fffff88003195230
        Base fffff88003196000 Limit fffff88003190000 Call 0
        Priority 13 BasePriority 12 UnusualBoost 0 ForegroundBoost 0
IoPriority 2 PagePriority 5
        Child-SP          RetAddr           Call Site
        fffff880`03194fc0 fffff800`02db4c57 nt!ObLogSecurityDescriptor+0x50
        fffff880`03195030 fffff800`02db6057 nt!SeDefaultObjectMethod+0x57
        fffff880`03195080 fffff800`02db4ee2 nt!ObpAssignSecurity+0xc7
        fffff880`031950f0 fffff800`02db76ff nt!ObInsertObjectEx+0x1e2
        fffff880`03195340 fffff800`02db6b06 nt!PspInsertThread+0x2f3
        fffff880`031954c0 fffff800`02d65da5 nt!PspCreateThread+0x246
        fffff880`03195740 fffff880`0799b3e2 nt!PsCreateSystemThread+0x125
        fffff880`03195830 fffff880`0799b6d8
Driver!SetupReadWorkQueue+0xe2
[x:\driver\isolate\workerqueue.cpp @ 125]
        fffff880`03195890 fffff880`0799c34a
Driver!SetupWorkerQueues+0x22c
[x:\driver\isolate\workerqueue.cpp @ 333]
        fffff880`03195970 fffff800`02eb32c7
Driver!DriverEntry+0x72 [x:\driver\isolate\driver.cpp @ 43]
        fffff880`031959a0 fffff800`02eb36c5 nt!IopLoadDriver+0xa07
        fffff880`03195c70 fffff800`02ad8641 nt!IopLoadUnloadDriver+0x55
        fffff880`03195cb0 fffff800`02d65e5a nt!ExpWorkerThread+0x111
        fffff880`03195d40 fffff800`02abfd26 nt!PspSystemThreadStartup+0x5a
        fffff880`03195d80 00000000`00000000 nt!KiStartSystemThread+0x16
```

This is the **driver entry** thread. It is actually setting up various global resources – in this case it is in the middle of creating a work queue for a custom queue package that runs in this driver. Here is the second thread:

```
     THREAD fffffa80018b9b50  Cid 0004.0038  Teb: 0000000000000000
Win32Thread: 0000000000000000 RUNNING on processor 1
        Not impersonating
        DeviceMap                 fffff8a0000060f0
        Owning Process            fffffa8001844840       Image:         System
        Attached Process          N/A            Image:         N/A
        Wait Start TickCount      264493         Ticks: 1 (0:00:00:00.015)
        Context Switch Count      52067          IdealProcessor: 2
        UserTime                  00:00:00.000
        KernelTime                00:00:01.357
        Win32 Start Address nt!ExpWorkerThread (0xfffff80002ad8530)
        Stack Init fffff88003180db0 Current fffff880031809e0
        Base fffff88003181000 Limit fffff8800317b000 Call 0
        Priority 13 BasePriority 12 UnusualBoost 1 ForegroundBoost 0
IoPriority 2 PagePriority 5
        Child-SP          RetAddr           Call Site
        fffff880`0317f7e8 fffff800`02e391c4 nt!KeBugCheckEx
        fffff880`0317f7f0 fffff800`02df405d
nt!PspUnhandledExceptionInSystemThread+0x24
        fffff880`0317f830 fffff800`02afa06c nt! ?? ::NNGAKEGL::`string'+0x227d
        fffff880`0317f860 fffff800`02af9aed nt!_C_specific_handler+0x8c
        fffff880`0317f8d0 fffff800`02af88c5
nt!RtlpExecuteHandlerForException+0xd
        fffff880`0317f900 fffff800`02b09851 nt!RtlDispatchException+0x415
        fffff880`0317ffe0 fffff800`02ace642 nt!KiDispatchException+0x135
        fffff880`03180680 fffff800`02acd1ba nt!KiExceptionDispatch+0xc2
        fffff880`03180860 fffff880`0799b724 nt!KiPageFault+0x23a
(TrapFrame @ fffff880`03180860)
        fffff880`031809f0 fffff880`0799c477
Driver! StopWorkerQueues+0x14
[x:\driver\isolate\workerqueue.cpp @ 351]
        fffff880`03180a20 fffff880`010fae09
Driver!UnloadCallback+0xd3 [x:\driver\isolate\driver.cpp
@ 76]
        fffff880`03180a80 fffff880`010f9dcd fltmgr!FltpDoUnloadFilter+0xf9
        fffff880`03180c70 fffff800`02ad8641 fltmgr!FltpSyncOpWorker+0x2d
        fffff880`03180cb0 fffff800`02d65e5a nt!ExpWorkerThread+0x111
        fffff880`03180d40 fffff800`02abfd26 nt!PspSystemThreadStartup+0x5a
        fffff880`03180d80 00000000`00000000 nt!KiStartSystemThread+0x16
```

This is a thread that is **unloading** the driver.

Upon seeing this we note that the driver load and unload are supposed to be serialized against one another **by the operating system**, as there is no way for a driver to protect against this scenario. It really does require external serialization to properly prevent this.

We did a bit of research and confirmed with our friends in Redmond that this problem is a known issue – and fixed in Windows 8. Unfortunately the system under test (and the customer solution itself) still requires support for Windows XP as the primary platform, and Windows 7 as the secondary platform. Windows 8 is not even on the customer's radar yet.

Solutions to Consider

One approach to handling this issue pre-Windows 8, could involve building a multi-driver system. The first driver would be responsible for starting the second driver in a serialized fashion. Driver 1 would load Driver2 via **ZwLoadDriver**. When this function returned successfully, Driver 1 would then call Driver 2 (via an IOCTL, FSCTL or export function) to actually perform the registration as a mini-filter.

Driver 2's Unload routine would call back to Driver 1 to ensure that the registration call had completed successfully by serializing with an EVENT object in Driver 1. Thus, this would ensure strict correct ordering between the two. The only purpose for Driver 1 would be to avoid this narrow race condition.

Another potential approach that we considered was to have the DriverEntry function create a device object. In the Unload routine, we can look at the **Flags** field of the device object to see if the **DO_DEVICE_INITIALIZING** bit has been cleared. If it has not, then we know that there is still a risk that **DriverEntry** has not yet exited and we should sleep and then check again.

(CONTINUED ON PAGE 13)

Load or Unload… (Cont.)

(CONTINUED FROM PAGE 8)

This relies upon the fact that the I/O Manager actually clears this bit after DriverEntry returns.

> ***Note*** *It is not necessary to clear the DO_DEVICE_INITIALIZING flag on device objects that are created in DriverEntry, because this is done automatically by the I/O Manager. However, your driver should clear this flag on all other device objects that it creates.*

Source: http://msdn.microsoft.com/en-ca/library/windows/hardware/ff539265(v=vs.85).aspx (Last Accessed August 2, 2013.)

Mitigation

Building a two driver system to protect against a very narrow race condition might be overkill in a situation like this. So rather than an outright solution, what can we do to at least minimize the window in which DriverEntry could still be running?

The simplest thing we can do is make sure the driver does filter registration as its last step – after setting up all of its other internal data structures and queues. This doesn't entirely prevent the crash, but it minimizes the window even further. This is ultimately the approach the owner of the driver took to solve the problem.

However, if that hadn't been enough, we proposed using a global driver event and set it at the end of DriverEntry. Then have the Unload wait on that event and afterwards pause for some period of time. This wouldn't entirely prevent the race condition but at least it would further minimize the window in which it could occur. Thus, a short (few seconds) delay is likely to be sufficient in most production environments.

Conclusions

Since observing this particular crash, we have followed this structure for our own mini-filters: we do registration at the end of our Driver Entry function. By doing so, we minimize the likelihood of the crash happening.

We have not explored the potential solutions or mitigations that we proposed, but we offer them to our readers for consideration in the event they need to further mitigate against this problem.

Follow us!

Death to Drive Letters
Drive Letter Alternatives

One of the challenges in building file system filter drivers is differentiating between the name space as typical applications see it to the version seen by drivers, particularly file system mini-filter drivers. A technique that we have used recently to aid in this involves using volume GUIDs rather than drive letters for local volumes. In this article, we will discuss why you might not want to use drive letters and describe how volume GUIDs can be used for local drives to provide consistent naming.

What's Wrong with Drive Letters?

The biggest problem with drive letters in a file system mini-filter is that we don't see them and when we do we aren't really sure what they mean.

Operations sent to local drives don't show us the drive letters used by the application. This is because they are symbolic links to the media volume on which the file system instance is mounted. By the time the name is stored in the file object passed to IRP_MJ_CREATE the device level naming, including that drive letter information, has been consumed by the Object Manager.

In fact, local drives do not even require a drive letter, which happens in the case of mount points. Further, in the case of mount points, the name the filter will see is just the name relative to the final volume – not to the original path. Thus, the path the application uses might be "\\?\c:\mountpoint\subdir" but what the filter will see this name (via IRP_MJ_CREATE) twice: first when it sees the original path (device object + "\mountpoint\subdir") and once again when it sees the reparsed path (second device object + "\subdir"). The first will complete with a STATUS_REPARSE return value. The second will complete based upon the existence of the object on the second volume. This significantly complicates reconstructing the name within a mini-filter driver if the expectation is that it will match the "original application name".

The reason this is complicated is that we frequently see rules driven mechanisms that are drive letter based – "let's intercept any file that is on the C drive with a *.docx suffix".

The complication then is how to determine when something is on the C drive from the filter. Since the drive letters are not provided to the filter, we have no easy way to determine when this is a match. There are functions (**IoQueryFileDosDeviceName**) that can be used in some circumstances, but they don't work with network volumes or when there is no drive letter. It also only returns one possible drive letter, even if there is more than one assigned.

Volume GUIDs

The alternative approach that we have used successfully in the past relates to the use of *Volume GUIDs*. These are unique identifiers associated with the given volume. The Mount Manager is a kernel mode driver that handles associating drive letters with volumes and it uses volume GUIDs for this task.

Drive letter assignment to physical devices is managed by the Mount Manager. If we look in the registry (HKLM\System\MountedDevices) we can see the current drive letter mappings known by the Mount Manager. An example of this can be seen in **Figure 1** (p. 15).

This information in the registry consists of the information used by the Mount Manager to map a given physical device to a corresponding drive letter **and** to its volume GUID name. In this way, changing the connection of the drive to the system does not change its drive letter. Note that not all drive letters present in that table are necessarily in use currently. This can happen, for example, if a disk with a partition table has previously been seen by the system and a persistent drive letter assigned to it.

To see the current list of volumes and their matching drive letters on your system, you can use the **mountvol.exe** utility, which is included in Windows. With no arguments, it will give you a list of all the current volumes and if they do have drive letters and/or mount points it will display that information.

You might also notice that each of those drives displays a *Volume GUID* based name. It turns out that you can actually use those volume GUID paths programmatically and in some of the UI components. These are persistent as well and are defined for all physical media volumes. Unfortunately, they aren't available for network drives, so we'll still have to do more work to handle the network case.

But let's look at how to manage volume GUIDs.

(CONTINUED ON PAGE 11)

Drive Letter Alternatives (Cont.)

(CONTINUED FROM PAGE 14)

Obtaining Volume GUIDs in Kernel Mode

Filter Manager provides a simple API for obtaining the volume GUID name **FltGetVolumeGuidName**. This function takes a caller provided buffer, a Filter Manager volume pointer (**FLT_VOLUME**) and an optional value indicating the buffer size that is required.

The code fragment in **Figure 1** demonstrates one way to obtain this information in kernel mode (Click link to see code in its entirety).

Note the use of the function **RtlGUIDFromString** in this code sample – a clear demonstration of how important GUIDs actually are in Windows – supported even in kernel mode as part of the OS runtime library.

One way we have used this in our own drivers is to collect this Volume GUID when we first see the volume – while setting up the volume context – and then we store it in the respective context structure. This allows us to easily grab that information later when we need it, and volume GUIDs won't normally change.

With this information we can now pass the volume GUID to our user mode service components.

Obtaining Volume GUIDs in User Mode

Obtaining a volume GUID in user mode is different and relies on using functions from the RPC libraries. **Figure 2** shows our sample function for obtaining a volume GUID (P. 16; Click link to see code in its entirety).

In this case our sample function takes a user mode path and figures out the volume GUID name of the volume where that path resides. If there is a mount point, this will resolve the mount point and return the volume GUID where the path or file resides.

This mechanism provides a clean and unambiguous way of telling a kernel mode driver which volumes are actually of interest, rather than using drive letters and/or mount points.

Volume GUIDs, Drive Letters and Mount Points

In our experience, the primary benefit to using Volume GUIDs rather than drive letters is they are unambiguous – there is only **one** volume with a given GUID, while a given drive can have zero or more drive letters – check out the **subst** and **assign** commands for examples of creating further aliases to existing drives and subdirectories.

(CONTINUED ON PAGE 16)

```
status = STATUS_SUCCESS;

    //
    // We use a while loop for cleanup
    //
    while (STATUS_SUCCESS == status)  {

        //
        // First call is to get the correct size
        //
        volumeContext->VolumeGuidString.Buffer = NULL;
        volumeContext->VolumeGuidString.Length = 0;
        volumeContext->VolumeGuidString.MaximumLength = 0;

        (void) FltGetVolumeGuidName(FltObjects->Volume, &volumeContext->VolumeGuidString, &bytesRequired);

        //
        // Let's allocate space
        //
        volumeContext->VolumeGuidString.Buffer = (PWCHAR) ExAllocatePoolWithTag (
                                                    PagedPool, bytesRequired, PRODUCT_MEMTAG_VOL_GUID);
        volumeContext->VolumeGuidString.Length = 0;
        ASSERT(bytesRequired <= UNICODE_STRING_MAX_BYTES);
        volumeContext->VolumeGuidString.MaximumLength = (USHORT) bytesRequired;

        if (NULL == volumeContext->VolumeGuidString.Buffer) {
            status = STATUS_INSUFFICIENT_RESOURCES;
            break;
        }

        //
        // Lets call it again
        //
        status = FltGetVolumeGuidName(FltObjects->Volume, &volumeContext->VolumeGuidString, &bytesRequired);
```

Click to Expand

Figure 1

Drive Letter Alternatives (Cont.)

(CONTINUED FROM PAGE 15)

Further, Volume GUIDs are persistent and drive letters can change. Thus, if the original policy is based upon drive letters, the user mode component should monitor changes to drive letters. This could be done using the WM_DEVICECHANGE notification, for example.

In addition there can be confusion about mount points. If someone specifies a pattern like "C:*" your user mode components will need to define what this means with respect to mount points. If it means "everything **including mount points**" then you will need to enumerate the mount points and determine if they are mounted on the C drive. This is typically done by using the Win32 function **FindFirstVolumeMountPoint**. Thus, you can use the volume GUID for the C: drive and then scan all the mount points on that volume.

Another alternative is to simply note that matching does not traverse across volume mount points – we note that directory change notifications don't cross volume mount points either, so there is some precedent for this behavior.

Regardless of which behavior you choose, be consistent and document it for your users to understand.

Conclusions

In our work, we've found that working with volume GUIDs simplifies a file system mini-filter considerably because it eliminates string handling and drive letter understanding. By moving that logic into user mode components we can simplify our mini-filter.

In our experience, any code we can move out of kernel mode is generally beneficial as it simplifies the driver. Further, string handling code is some of the most sensitive and error prone code in a kernel mode driver, so this leads to a more robust solution.

```
//
// GetVolumeGuid
//
//  The purpose of this function is to extract the volume GUID
//  for the given file. Note that this will extract current
//  volume/path from the current context.
//
// Inputs:
//  OriginalFilePathName - this is the file and path to check
//
//
_Success_(return) BOOLEAN GetVolumeGuid(_In_z_ TCHAR *OriginalFilePathName, __out GUID *Guid)
{
    TCHAR *filePathName;
    ULONG filePathNameSize = UNICODE_STRING_MAX_BYTES;
    TCHAR guidVolumeName[64]; // these names are fixed size and much smaller than this
    USHORT index;
    RPC_STATUS rstatus;
    TCHAR *fileNamePart;

    filePathName = (TCHAR *) ExAllocatePoolWithTag(PagedPool, filePathNameSize, POOL_TAG_FILE_NAME_BUFFER);

    if (NULL == filePathName) {
        return FALSE;
    }

    filePathNameSize /= sizeof(TCHAR);

    GetFullPathName(OriginalFilePathName,filePathNameSize,filePathName,&fileNamePart);

    //
    // We now have a path name, let's see if we can trim it until we find a valid path
    //
    index = (USHORT) _tcslen(filePathName);

    if (0 == index) {
```


Figure 2

Using Bus Interfaces... (Cont.)

(CONTINUED FROM PAGE 7)

Size – The overall size of the structure, including the driver-defined portions

Version – A version for the structure. The consumer of the interface requests which version of the structure they would like, thus it's possible to have multiple versions of the same interface

Context – A per-interface instance context value for the producer

InterfaceReference – A producer provided routine to acquire a reference to the interface

InterfaceDereference – A producer provided routine to release a reference on the interface

While most of those members are self-explanatory, the *InterfaceReference* and *InterfaceDereference* members require a bit of explanation.

According to the documentation, any time the producer exports the interface to a consumer, the *InterfaceReference* routine must be called by the producer. Likewise, once the consumer is done with the interface, the consumer must call the *InterfaceDereference* routine.

These referencing and dereferencing callbacks can serve two purposes. One is that the producer may have some per-interface instance state that must be torn down when the consumer is finished. Imagine a scenario where the producer allocates memory each time an interface is returned to a consumer. In this case, the producer must free the memory when the reference count on the interface goes to zero.

The other purpose of these callbacks is for the case of a consumer that is using an interface provided by a device other than their own PDO. Remember that in these cases we have to worry about potential teardown issues, as the I/O Manager does not happen to prevent the producer from being removed before the consumer. In these cases, these callbacks may be leveraged to prevent the producer from unloading until the last consumer releases its reference. This can unfortunately be a bit trickier than it sounds. The producer must be careful to not trigger its own unload from directly within the dereference callback. If it does, the producer runs the risk of unmapping the code for the dereference callback while it is still executing within the callback.

For most drivers, the tearing down of the interface does not require the producer to undo any state. If possible, it can be easier to solve the cross stack teardown problems in the consumer (e.g. by maintaining an open File Object to the producer's Device Object). Thus, for many drivers the *InterfaceDereference* routine becomes an empty routine that does nothing. However, the presence of both the reference and dereference routines is still required by the O/S. Thankfully, the Framework developers anticipated this situation and provide default "no op" routines for your driver to use for both *InterfaceRereference* (**WdfDeviceInterfaceReferenceNoOp**) and *InterfaceDereference* (**WdfDeviceInterfaceDereferenceNoOp**).

We can now see all of the steps involved in being a Bus Interface provider in a KMDF driver:

1. Define a GUID for your Bus Interface
2. Define our own custom INTERFACE structure
3. Initialize **WDF_QUERY_INTERFACE_CONFIG** structure with **WDF_QUERY_INTERFACE_CONFIG_INIT**
4. Call **WdfDeviceAddQueryInterface** to associate the interface with our device

The Framework then takes care of absolutely everything else. Pretty cool, eh? Let's now see these steps in action.

Producer Step 1: Define a GUID for your Bus Interface

Nothing too shocking here, we define a GUID using the standard **DEFINE_GUID** macro provided with the WDK (**Figure 7**). Just

(CONTINUED ON PAGE 18)

Using Bus Interfaces... (Cont.)

(CONTINUED FROM PAGE 17)
don't forget to include *initguid.h*!

```
// {9671F9BD-F7A7-495c-AA84-74FEBCD07934}
DEFINE_GUID(GUID_NV2BUDDY_BUS_INTERFACE,
0x9671f9bd, 0xf7a7, 0x495c, 0xaa, 0x84, 0x74, 0xfe, 0xbc,
    0xd0, 0x79, 0x34);
```

Figure 7

Producer Step 2: Define a custom INTERFACE structure

In our driver, we'd like to let other drivers directly call a routine to write to our device. Thus, for version one of our Bus Interface we'll provide a single write routine. Our driver entirely controls the function prototype of our write routine, thus we can require the consumer to pass any parameters that we wish. In our case, we'll be sure to have the consumer pass the **INTERFACE** structure back to us as the first parameter. This allows us to retrieve the **Context** member that we supplied the consumer when they queried for the Bus Interface. The full definition of our custom interface structure can be seen in **Figure 8**.

```
typedef
NTSTATUS
(*PNV2BUDDY_WRITE)(
    _In_ PINTERFACE InterfaceHeader,
    _In_ PVOID WriteBuffer,
    _In_ size_t WriteBufferLength,
    _Out_ size_t *BytesWritten
    );

typedef struct _NV2BUDDY_BUS_INTERFACE {
    //
    // Standard interface header, must be present
    //
    INTERFACE       InterfaceHeader;

    //
    // Our driver supplied routines
    //
    PNV2BUDDY_WRITE Nv2BuddyWrite;

}NV2BUDDY_BUS_INTERFACE, *PNV2BUDDY_BUS_INTERFACE;

#define NV2BUDDY_BUS_INTERFACE_VERSION      1
```

Figure 8

*Producer Step 3: Initialize a **WDF_QUERY_INTERFACE_CONFIG** structure*

We initialize our structure by calling **WDF_QUERY_INTERFACE_CONFIG_INIT**, which requires a pointer to our interface structure as well as our interface GUID. We already have our interface GUID defined in a header, so we just need to define and initialize a **NV2BUDDY_BUS_INTERFACE** structure.

Note that initializing this structure comes in two phases. First, we'll initialize the common, O/S-defined header. For our interface's **Context** member we'll supply a handle to our WDFDEVICE object. We'll also use the WDF-supplied dummy routines for our reference counting needs (**Figure 9**).

```
NV2BUDDY_BUS_INTERFACE busInterface;
PINTERFACE             interfaceHeader;

  //
  // Set up the common interface header
  //
  interfaceHeader = &busInterface.InterfaceHeader;

  interfaceHeader->Size    = sizeof
                    (NV2BUDDY_BUS_INTERFACE);
  interfaceHeader->Version =
                    NV2BUDDY_BUS_INTERFACE_VERSION;
  interfaceHeader->Context = (PVOID)device;

  //
  // We don't pay any particular attention to the
  // reference counting of this interface, but we MUST
  // specify routines for it. Luckily the framework
  // provides dummy routines
  //
  interfaceHeader->InterfaceReference   =
                    WdfDeviceInterfaceReferenceNoOp;
  interfaceHeader->InterfaceDereference =
                    WdfDeviceInterfaceDereferenceNoOp;
```

Figure 9

Next, we'll initialize the driver-specific portion of the interface structure. For our this we'll simply fill in the write routine to be an existing routine in our driver (**Figure 10**).

```
  //
  // Now we can fill in our bus interface callbacks
  //
  busInterface.Nv2BuddyWrite = Nv2BuddyInterfaceWrite;
```

Figure 10

We're now ready to initialize a **WDF_QUERY_INTERFACE_CONFIG** structure with **WDF_QUERY_INTERFACE_CONFIG_INIT** (**Figure 11**).

```
WDF_QUERY_INTERFACE_CONFIG queryInterfaceConfig;

WDF_QUERY_INTERFACE_CONFIG_INIT(&queryInterfaceConfig,
                                interfaceHeader,
                                &GUID_NV2BUDDY_BUS_INTERFACE,
                                WDF_NO_EVENT_CALLBACK);
```

Figure 11

*Producer Step 4: Call **WdfDeviceAddQueryInterface***

Now for the easy part. We simply need to call **WdfDeviceAddQueryInterface** to associate the interface with our WDFDEVICE. Again, remember that the **type** of device does not matter, it could be a function, filter, or physical device (**Figure 12**).

(CONTINUED ON PAGE 19)

Using Bus Interfaces... (Cont.)

(CONTINUED FROM PAGE 18)

```
    //
    // Add the interface!
    //
    status = WdfDeviceAddQueryInterface(device,

    &queryInterfaceConfig);

    if (!NT_SUCCESS(status)) {
#if DBG
        DbgPrint(
               "WdfDeviceAddQueryInterface failed 0x%0x\n",
                status);
#endif
        return(status);
    }
```

Figure 12

Consuming a Bus Interface in KMDF

On to the consumer! For the consumer, we ask the Framework to query an interface for us. When we do this we can either query for an interface *within our own stack* with **WdfFdoQueryForInterface** or within a *different* stack with **WdfIoTargetQueryForInterface**.

The **WdfFdoQueryForInterface** method (**Figure 13**) queries the current device stack for an interface that we identify via GUID. The interface query is sent to the top of the device stack, thus it's possible to retrieve a Bus Interface of a device above the given device.

```
_Must_inspect_result_
_IRQL_requires_max_(PASSIVE_LEVEL)
NTSTATUS
WdfFdoQueryForInterface(
    _In_ WDFDEVICE Fdo,
    _In_ LPCGUID InterfaceType,
    _Out_ PINTERFACE Interface,
    _In_ USHORT Size,
    _In_ USHORT Version,
    _In_opt_ PVOID InterfaceSpecificData
    );
```

Figure 13

The **WdfIoTargetQueryForInterface** method (**Figure 14**) performs the exact same operation, though, as the name implies, it requires a handle to a Remote I/O Target representing a device in a different device stack.

```
_Must_inspect_result_
_IRQL_requires_max_(PASSIVE_LEVEL)
NTSTATUS
WdfIoTargetQueryForInterface(
    _In_ WDFIOTARGET IoTarget,
    _In_ LPCGUID InterfaceType,
    _Out_ PINTERFACE Interface,
    _In_ USHORT Size,
    _In_ USHORT Version,
    _In_opt_ PVOID InterfaceSpecificData
    );
```

Figure 14

Each of these APIs requires a size and version for the Bus Interface to be queried, which the Framework will use to validate that the consumer is requesting a version of the interface that is supported by the producer. They also take an optional *InterfaceSpecificData* parameter that can be used to pass information about the requested interface to the producer. For most Bus Interfaces this member will be NULL, but it may be something that could be useful in your drivers. Note that the producer must register an *EvtDeviceProcessQueryInterface Request* event processing callback to receive this parameter.

Once the consumer has successfully called either of these APIs, it can begin using the returned interface structure to call into

(CONTINUED ON PAGE 20)

Using Bus Interfaces... (Cont.)

(CONTINUED FROM PAGE 19)

the producer. We can see an example of querying and using a remote Bus Interface in **Figure 15**.

```
    NV2BUDDY_BUS_INTERFACE  buddyBusInterface;
    status = WdfIoTargetQueryForInterface(
                              DevContext->BuddyTarget,
                              &GUID_NV2BUDDY_BUS_INTERFACE,
                              (PINTERFACE)&buddyBusInterface,
                              sizeof(NV2BUDDY_BUS_INTERFACE),
                              NV2BUDDY_BUS_INTERFACE_VERSION,
                              NULL);

    if (!NT_SUCCESS(status)){

#if DBG
        DbgPrint("WdfIoTargetQueryForInterface failed 0x%0x\n", status);
#endif

        return(status);

    }

    //
    // Call the write routine!
    //
    status = (*buddyBusInterface.Nv2BuddyWrite)(
                              &buddyBusInterface.InterfaceHeader,
                              inputBuffer,
                              inputBufferLength,
                              &bytesWritten);
```

Figure 15

When the consumer has finished using the interface, it must be sure to call the *InterfaceDereference* member of the returned **INTERFACE** structure. This is the signal to the producer that the consumer has finished using the interface. As a debugging aid, it's recommended to zero the interface structure after it has been released. This will trigger an immediate bugcheck if the consumer inadvertently uses the interface after it has been released. **Figure 16** demonstrates releasing the interface.

```
    PNV2BUDDY_BUS_INTERFACE buddyBusInterface;
    PINTERFACE              interfaceHeader;

    buddyBusInterface = &DevContext->BuddyBusInterface;

    interfaceHeader = &buddyBusInterface->InterfaceHeader;

    // Deref the interface!
    //
    (*interfaceHeader->InterfaceDereference)
                              (interfaceHeader->Context);

    // And zero it for debugging purposes
    //
    RtlZeroMemory(buddyBusInterface,
                  sizeof(NV2BUDDY_BUS_INTERFACE));
```

Figure 16

If the consumer is taking on the responsibility for ensuring the producer does not unload, at this point the consumer might also release their reference on the producer (e.g. by closing a Remote I/O Target to the device).

Bi-Directional Bus Interfaces

Up to this point we've only been discussing a strict producer and consumer relationship: the producer provides a structure, the consumer retrieves the structure and calls into the producer. If you're using WDF, by default this is the only behavior you can achieve. The Framework will copy the producer's custom interface structure into the consumer's output buffer, resulting in the consumer receiving a private copy of the interface. The contents of the consumer's buffer are ignored on input, and any subsequent modifications made to the buffer are seen only by the consumer.

To override this behavior, the producer may set the *ImportInterface* member of the **WDF_QUERY_INTERFACE_ CONFIG** structure to TRUE. That's a little strange, but what it means is the producer may now *import* members of the custom interface buffer that the consumer provides. If this member is TRUE, the producer must also specify an *EvtDeviceProcessQuery InterfaceRequest* event processing callback to receive the consumer's buffer. During this callback the producer may read from the consumer's interface structure, write to the consumer's interface structure, or both.

Conclusion

We continue to find Bus Interfaces useful here at OSR and hopefully this article convinced you to take a first (or second) look. The WDF support makes them easy to produce and consume, though please keep in mind the issue of teardown if you're not simply using an interface provided by your PDO.

The WDK Docs Improve... (Cont.)

(CONTINUED FROM PAGE 9)

Tail.Overlay.Thread

A pointer to the caller's thread control block (TCB). For requests that originate in user-mode, the I/O manager always sets this field to point to the TCB of the thread that issued the request.

Well, **that's** a metric duckload better, don't you think? Heck, **that** definition is actually useful. And accurate. This is just one example of the kind of cleanup and clarification that's taking place throughout the WDK.

I won't even mention how great it is to have a member of the doc team who regularly reads and contributes to the NTDEV forum (yay for Diane Olsen!). She's quick to jump into discussions, take bug reports, and even keep us informed of major doc additions and changes.

The Best Yet

I'm not saying the WDK docs are perfect. There's still room for improvement, of course. For example, the docs could give us more information on the error codes returned from functions and what conditions cause those errors, for example. The function call examples that appear on the doc pages could be (cough, cough) a bit less trivial than they sometimes are today. And, I will **always** want more architecture information… more details about design motivations, tradeoffs, and side-effects.

But if you stop for a moment and take an overall look at the latest WDK documentation, I think you'll be both surprised and pleased at what you see. The changes really are impressive. I have no trouble saying the WDK docs are in the best state they've ever been in at any time in the history of Windows.

Community Interaction

What do you think about the state of the WDK docs? Tweet us @osrdrivers and use #TheNTInsider to provide your comments. Don't tweet? C'mon...even Peter tweets. Well, there's always Facebook and LinkedIn...or the telegraph...

Follow us! 

Understanding EvtIoStop… (Cont.)

(CONTINUED FROM PAGE 11)

In general, the way to avoid the dreaded bugcheck 0x9F is to be sure you implement an EvtIoStop Event Processing Callback for each of your Queues. Doing this is simple. You just specify your callback as part of your WDF_IO_QUEUE_CONFIG structure as shown in **Figure 2**.

```
    //
    // Create our default queue for I/O requests.
    //
    WDF_IO_QUEUE_CONFIG_INIT_DEFAULT_QUEUE(&ioQueueConfig,
                                           WdfIoQueueDispatchParallel);

    ioQueueConfig.EvtIoDeviceControl = FooEvtFdoDeviceControl;
    ioQueueConfig.EvtIoStop          = FooEvtIoQueueStop;

    status = WdfIoQueueCreate(device,
                              &ioQueueConfig,
                              WDF_NO_OBJECT_ATTRIBUTES,
                              WDF_NO_HANDLE);

    if (!NT_SUCCESS(status)) {
#if DBG
        DbgPrint("WdfIoQueueCreate failed status 0x%x", status);
#endif
        return status;
    }
```

Figure 2—Specifying an EvtIoStop Callback

EvtIoStop is Optional

Do you always need to specify an **EvtIoStop** Event Processing Callback in your driver? The answer is no. While it is never a mistake to do so, there are cases when don't need to implement **EvtIoStop**. This is where the long and obtuse comment and **_Analysis_assume** in the WDK samples comes in. If your driver:

- Completes every Request it receives synchronously – that is, it never holds any Request in progress, or
- Receives Requests from a WDF Queue that is not power managed, or
- Guarantees that **every** Request that will be in progress when a device is asked to transition to a lower-powered D-State will **always** complete quickly…

…then your driver does **not** need to supply an **EvtIoStop** Event Processing Callback.

The most controversial of the above cases is the last one: When your driver can guarantee that every in-progress Request will complete quickly. The central issue here is what "quickly" should mean. In this case, we mean **really** fast... probably less than a second. Remember: The system might be waiting for your device to complete its transition to a lower-powered D-State, and for that to happen your active Requests have to complete. While the system provides for a lot longer timeout than one second before generating the 0x9F bug check, the goal is to provide a good user experience when the system is suspending or hibernating. Keeping your Request completion times short helps with this.

By default, SDV requires that every driver that utilizes a power managed Queue provide an EvtIoStop Event Processing Callback for that Queue. If you want your driver to pass SDV clean, with no warnings (and you **should**) then you'll need to stop SDV from complaining. The easiest way to do this is via the **_Analysis_assume** shown in **Figure 1**. What this does is tell SDV to just assume that we have specified an **EvtIoStop** Event Processing Callback… even though we haven't.

Power, Queues, and Requests

So that's the story of how device power state transitions are related to WDF Queues, and how the need to stop WDF Queues relates to having an **EvtIoStop** Event Processing Callback in your driver. This is also an example of how requirements, and what's considered "best practice" among developers, change over the years. The traditional method of handling Requests during transitions to a lower power state was "just let them finish and otherwise don't worry about it." This was true even in the earlier days of WDF. But as more emphasis is placed on battery performance, portable systems often sleep a lot more frequently. In this case "just let them finish" can be a bad policy. Not to mention, losing I/O requests across power state transitions wasn't ever a good thing.

With a properly considered policy, and by implementing **EvtIoStop** when required, your driver will be more likely to handle device power state transitions properly.

Peter Pontificates... (Cont.)

(CONTINUED FROM PAGE 5)

OK, so a lot of **old** high tech came from here. That doesn't mean the Boston area sucks. There's tons of **new** tech happening in the Boston area, too. Microsoft has a research lab in Cambridge. There's something of a growing community of speech recognition specialists in the Cambridge area as well. There's actually a special concentration of expertise in both storage and system virtualization located in the Boston area: Oracle, EMC, Citrix. I'm sure I'm missing a few. Virtual Computer, which was bought by Citrix and made a lot of folks a ton of money, started here. Heck, Redhat even has a development team here.

Bottom line: Despite what people tend to think, they do not have to live in Silicon Valley. There's plenty of tech here in the Boston area. Even tech for kernel-mode devs. In Both Windows and Linux.

Real estate is cheap compared to both the Valley and the Seattle area, and the quality of life is amazing. Art, music, universities... quick access to what we on the East Coast consider skiing, awesome beaches. All in a background steeped in American History – think Pilgrims and Paul Revere's ride.

We like New England. It's great here. Now we just need to convince a few engineers that this is a fun place to be.

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.

OSR Seminar Schedule

Seminar	Dates	Location
Internals & Software Drivers	23-27 June	Dulles/Sterling, VA
Advanced WDF Drivers	14-17 July	Boston/Waltham, MA
Kernel Debugging & Crash Analysis	18-22 August	Palo Alto, CA
WDF Drivers	22-26 September	Boston/Waltham, MA
Developing File Systems	4-7 November	Seattle, WA

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...below are some results from recent surveys of attendees of OSR seminars:

- [Instructor] was fantastic and I can easily say that this has been one of the best training seminars I have attended.
- [Instructor] was an awesome trainer. I wish to attend more training sessions in the future.
- Everything I expect from OSR, which is a high standard.
- I was VERY impressed with the content and the instructor's knowledge of the subject matter. All questions were answered for all students and/or researched quickly, if an answer was not readily available. In my post-trip report, I have already recommended that more personnel from our office attend this course.
- The seminar was great. Even with previous knowledge on WDF drivers, I left the seminar feeling like I learned a bunch of new concepts.
- It was a very interesting, fast-paced, introduction to the development of Windows file system drivers. The instructor was very knowledgeable and experienced, bringing various examples from real world applications into the classroom.
- This was a good learning experience for me to enrich my Windows knowledge base.
- It was well run and covered a lot of good material. [Instructor] is obviously very knowledgeable and presents the material in an enjoyable manner.
- The OSR seminar was a great learning experience for me. I am planning on attending another seminar early next year.
- Simply awesome. I am looking forward to attending more seminars from OSR.

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.