ISSUE VOLUME
01/20
JAN-FEB
2014

# The NT Insider

A publication of OSR Open Systems Resources, Inc.

```
NTSTATUS                  status;
WDF_INSIDER_ATTRIBUTES newestInsider;
WDF_ARTICLE_INFO          articles;

WDF_INSIDER_ATTRIBUTES_INIT(&newestInsider,
                            "January/February 14");

WDF_ARTICLE_INFO_INIT_SPECIFY_ARTICLE_COUNT_    \
           USING_CURRENT_TYPE_DEFAULT_QUEUE(&articles,
                                            6);
PAGED_CODE();

// Getting Started Writing Windows Drivers
articles.ArticleList[0].Title = "Ready, Set---Go!";
articles.ArticleList[0].Page  = 4;

// Seminar: Advanced WDF Driver Development
articles.ArticleList[1].Title = "OSR's Newest Seminar";
articles.ArticleList[1].Page  = 5;

// WDF File Object Callbacks Demystified
articles.ArticleList[2].Title = "Grand Opening";
articles.ArticleList[2].Page  = 6;

// Windows Pool Manager
articles.ArticleList[3].Title = "Debugger's Delight";
articles.ArticleList[3].Page  = 8;

// The Truth Behind PAGE_FAULT_IN_NONPAGED_AREA
articles.ArticleList[4].Title = "Analyst's Perspective";
articles.ArticleList[4].Page  = 10;

// Driver Installers
articles.ArticleList[5].Title = "Tim Roberts on Installation";
articles.ArticleList[5].Page  = 11;

articles.PontificationTitle = "Always Forward, Onward...";
articles.PontificationPage  = 3;

WdfInsiderAttributesSetArticleInfo(&newestInsider,
                                   &articles);

status = WdfInsiderCreate(&newestInsider,
                          WDF_NO_OBJECT_ATTRIBUTES,
                          WDF_NO_HANDLE);

if (!NT_SUCCESS(status)) {
    DbgPrint("Can't create article\n");
    // TODO: Is goto OK here??
    // goto exit;
    return(STATUS_UNSUCCESSFUL);
}
```

Other Special Features

OSR Social Media....P.2

Careers at OSR....P.23

Seminar Schedule....P.24

**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*

## Socializing
## OSR Social Media

When it comes to instituting new procedures, making decisions or picking up the latest tech, OSR's modus operandi tends to be "don't think...just run with it". Of course, there have been some advancements in the technology space that we prefer to sit back and watch, make fun of, insist will die a painful death, and then...slowly come around on.

The social media craze is one of them.

Thus, over the coming weeks and months, you can expect us to begin using our social media sites to provide timely technical insight, announce new initiatives or programs intended to assist the community, and generally update folks as to "goings on" at OSR.

Click one of the icons below to join the fun, or find us at:

**FB**: https://www.facebook.com/pages/OSR-Open-Systems-Resources-Inc/131083523584516
**Twitter**: http://www.twitter.com/osrdrivers
**LinkedIn**: http://www.linkedin.com/company/osr

### 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.

Whether you're looking for training, consulting, or somebody for the development of your project end-to-end… 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

## Peter Pontificates:
## Always Forward, Onward, To the Future! Now! Hurry! Let's Go!

Can you hear it? It's the drumbeat that says "forward, onward, upgrade to newer/faster/better/newer/newer/newer." There's no place this drum beat is louder than in Redmond, Washington.

I'm sure you've read that Windows XP (SP 3, to be specific) will be reaching the end of its super-duper double secret Extended Support as of April of this year. But did you know that XP has been out of what's called "Mainstream Support" since 2009? And, for both people who installed it, Mainstream Support for Windows Vista ended sometime in 2012. Heck even support for Windows 7 RTM (with no service packs) ended last year in 2013.

I **get** that support has to end sometime. All good things must come to an end. And I **get** that Windows XP RTM'ed in 2001, and XP SP3 was released in 2008, two years **after** Windows Vista.

Here's the problem, though: Lots of people are still running Windows XP. Lots and lots of people. How many? ZDNet (an authority second only to Wikipedia) says that between **25 and 33 percent of all the world's desktops** are still running Windows XP (http://www.zdnet.com/windows-xp-what-to-expect-once-microsoft-shuts-down-support-7000025348/). Why? Probably because XP works perfectly well for the purpose it's used and the environment that it's being used in. Heck, I've got a computer at home that I use as part of an amateur radio system that still runs XP. It runs very well, too, thank you very much.

You're probably wondering why I'm pontificating on this topic. I write system software. Who cares about XP, right? Let me answer that for you: Our clients care about Windows XP. And not because they have amateur radio setups that use it, either. Our clients care because **their customers care**. That means our clients make supporting Windows XP our problem.

So if OSR writes a driver in many cases (not all, mind you, but many) the client will want that driver to support Windows XP or Windows Server 2003. Heck, we just finished designing and implementing a complex piece of storage monitoring software that had to work on Server 03 and later.

But we've got KMDF and we can always just ship that latest co-installer and we're good to go, right? The most recent rev of KMDF is V1.13, so we can just use that. Well, wrong. This is where it gets sticky.

The Windows 8 WDK only supports building drivers targeted at Windows Vista and later. The Windows 8.1 WDK only supports building drivers for systems running Windows 7 or later. OK, so using the 8.1 WDK and losing Vista support is no big loss…who cares about Vista. The problem I've discovered is people DO still care about Server 2008.

So if I want support for Windows XP, I have to use the Windows 7 WDK. And the WDK for Windows 7 only supports KMDF V1.9. What's worse is that the Windows 7 WDK uses the old sources/dirs method of building driver projects. Starting in Windows 8, the WDK uses the new driver development environment that's integrated with Visual Studio and does not support the old sources/dir projects. So, if I need to support Windows XP I need

(CONTINUED ON PAGE 23)

KERNEL DEBUGGING & CRASH ANALYSIS SEMINAR

Fundamentals, Tips, Tricks & Techniques

Next presentation:

Dulles/Sterling, VA
24-28 March

Phone: +1.603.595.6500
Email: seminars@osr.com

DEVELOPING FILE SYSTEMS FOR WINDOWS SEMINAR

Let Industry-Recognized Experts Guide Your Learning

Next presentation:

Boston/Waltham, MA
13-16 May

Phone: +1.603.595.6500
Email: seminars@osr.com

# Ready, Set...Go!
# Getting Started Writing Windows Drivers

You're working at some company. They've decided to support a new device on Windows. Or perhaps they want to add some features to an existing device that's already supported by Windows. Or, just maybe your company needs to collect some information from the operating system, which is only available from kernel-mode. Regardless of which of these describes your situation, somebody is going to need to write a driver to accomplish this goal.

Since you're a clever guy and eager to learn new stuff, your managertroid asks you if you'd like to write the driver. You say "Sure!" The problem? You've never written a driver for Windows before. Where do you begin?

Believe it or not, one of the most commonly asked questions we receive here at OSR is "How do I write a driver for Windows?" You'd think the answer would be simple. And **sometimes** it is. But, all too often, the answer is not only non-obvious, it's fraught with complexity.

The answers to the question, "How do I write a driver for my device on Windows" come in three categories:

- What you need to know
- What development tools (and stuff) you need
- What driver model to use

We'll describe each of these in individual sections, below.

## What You Need To Know

The things you need to know fall into two categories:

- Personal background about the Windows operating system and devices that'll allow you to readily learn about how to write Windows drivers.

- Technical information about the hardware device you need to write your driver for (if you're writing a driver for a hardware device).

### *Personal Knowledge*

Items in the first category, personal background knowledge, are actually pretty simple. To be able to write drivers for Windows and not just confuse yourself, you need to have at least general knowledge of computer operating systems and Windows in particular. You probably know most of what you need if you took a general OS Concepts class when you were in school. If you understand about devices, registers, interrupts, virtual memory, scheduling, multi-threaded programming, reentrancy, and concurrency issues… you're more than half-way there. You can pick-up the Windows-specific information you need from doing a bit of reading. Please don't skip this step. We spend almost two days in the 5-day driver seminar we teach here at OSR just discussing Windows OS and I/O subsystem architecture. So, it's important.

Also, if you're not familiar with programming on Windows systems from a user perspective (maybe you've been working in Linux all your life… if so, first of all, I'm sorry… but I digress) it would also be helpful to know a bit about Windows I/O fundamentals.

If you need to brush-up on your OS concepts, would like to know more about Windows OS concepts in particular, or you'd like to learn more about how I/O is performed in Windows, we have some reading suggestions in the Sidebar labeled **Understanding Windows OS and I/O Concepts** (See P. 16). Doing that reading should set you up well for your task for writing Windows drivers.

One other thing you'll need to know in terms of personal background is something about the hardware architecture that's typical of the platform on which your hardware will be running. Whether the device you're writing the driver for will run on PC (desktop to server) systems or used exclusively in an ARM SoC system, knowing something about the hardware environment – such as common buses and hardware concepts – that are unique to that platform would be valuable. You don't need to know a lot. We're not saying you need to be a hardware designer. We're just saying knowing, for example, the basic concepts of PCIe or USB or SPI or whatever bus your device connects to will help speed you on your way as you write your driver.

### *About Your Hardware*

If you're writing a driver to support a hardware device on Windows, you'll need the hardware specifications for the device you'll be supporting. The information you need usually takes the form of a "data sheet" (which is often more like a book than a single sheet of paper) that describes the register-level interface to your device. Your hardware designer can give this to you. You need the **specifics** of your device, by the way. If the device you'll be writing your driver for is implemented using some sort of PLD like an FPGA, don't let your hardware designer simply point you off to the hardware spec for the PLD device (hardware designer waves her hand at you while saying: "Oh, we're using an Arria II GX. Just go to Altera's web site and download what you need. Bye."). You need to know how the designer has implemented the register interface using the chosen PLD device, not the specs for the PLD itself.

## Development Tools (and Stuff) You Need

Over the past few years, the tools used for Windows driver development have undergone nothing short of a revolution. Gone (well, mostly) are the days when you had to use special

(CONTINUED ON PAGE 16)

## OSR'S NEWEST SEMINAR: ADVANCED WDF DRIVER DEVELOPMENT

Over the past couple of years several of our clients and a bunch of our students have convinced us that there's a real need for folks to move beyond the basics of WDF. And if people are going to do this, then the least we can do is give them the benefit of our experience. After all, the folks here at OSR were involved in the design of WDF from its earliest days, and have been writing KMDF drivers since even before the model was first publically released.

In thinking about what should be in our **Advanced WDF Driver Development** class, we thought that – more than anything – the class should be focused on solving common challenges that people encounter using practical methods. By "practical" here, we mean both documented and well-understood, even if that understanding isn't common knowledge among WDF developers.

For example, many people want to learn about optimal methods of notifying applications of events and moving data between user-mode and kernel mode. Those of you who spend any time reading NTDEV will know that we most frequently recommend people simply use read/write requests, and for event notification we recommend the Inverted Call Model. We recommend these approaches because they work best for most times and for most uses. But suppose you need better performance or lower overhead than you get with Buffered I/O or Direct I/O?

There are indeed alternative techniques you can use. Some of those are Neither I/O, Fast I/O for Device Control, and shared memory. The problem with recommending these techniques is that they require a level of understanding that's considerably greater than that required to get a WDFREQUEST off a WDFQUEUE. It's not that they require some deep, secret, mystic, knowledge of the source code or that these techniques are hard. Rather, these techniques require a more considered approach, one in which the applicable conditions, their advantages, and disadvantages are all carefully evaluated. Learning to use these techniques so their benefits are maximized and their costs minimized, generally requires a bit of experience.

INAUGURAL PRESENTATION

Advanced WDF Drivers

Palo Alto, CA   5-8 May

Free OSR coffee mug to all attendees of this presentation!

Phone: +1.603.595.6500
Email: seminars@osr.com

We created our **Advanced WDF Driver Development** seminar with precisely these types of problems in mind. Whether it's kernel-mode to user-mode communication, communication between drivers, or debugging invalid memory accesses (when **do** you get PAGE_FAULT_IN_NONPAGED_AREA versus IRQL_NOT_LESS_OR_EQUAL? We'll tell you!), our approach is to describe the problem domain, present various possible solutions in order of increasing complexity, discuss the advantages and disadvantages of each potential solution, and then present some guidelines about when each solution might best be used.

In addition to these bigger issues, we also spend time on some of the smaller practical issues we all encounter in WDF driver development. Things like why relying on WDF object parenting for child object destruction isn't always as simple as it seems. Or what's causing all those 0x9F blue screens that people seem to be seeing lately. We even spend time talking about work queues, both system-provided and rolling your own, and when we think it makes sense to use each.

We also, finally, have the time in this seminar to discuss the topics that people have most frequently asked us about over the years: Busmaster DMA (including MSI, MSI-x and NUMA issues), writing bus drivers, and even our recommendations of how to use the wide variety of tools (SDV, Code Analysis, Windows DV) available to driver developers these days.

Check out the outline for the seminar. Maybe you'll join us for our inaugural offering? Everyone who attends our first public Advanced WDF Driver Development seminar will get a cool commemorative OSR coffee mug. See you there?

# Grand Opening
## WDF File Object Callbacks and Properties Demystified

Most drivers don't care when an application or a driver opens or closes their device. In fact, not caring is *so* common that you have to go out of your way in WDF to be notified of these operations. However, you might want to be notified in order to track the number of open instances on your device, or to keep track of per open instance state. For example, let's say that your driver provides some sort of encryption service. Each time a user opens your encryption device, you generate a unique key that'll be used by all I/O requests sent on that open instance.

Of course, any time you implement features that aren't commonly used, there's a chance for the documentation and samples to be a bit lacking. In this article, we'll cover all the options that are available to you if you find yourself in the not so common case.

### WDFFILEOBJECTs and KMDF

The native File Object represents a single, specific, open instance of a device (or a file on a device). Applications create new File Objects by calling **CreateFile** and destroy File Objects by calling **CloseHandle**.

The KMDF abstraction of the native File Object is the WDFFILEOBJECT. The WDF File Object closely mirrors the native File Object and is used to represent a unique open instance of a WDF Device Object. As with all WDF objects, the WDF File Object supports a series of Event Processing Callbacks, properties, and the standard Common Object Attributes.

*"In general, this is why it makes sense to defer tearing down any unique state for this open instance until the EvtFileClose Event Processing Callback. This eliminates any races between tearing down the open instance state and using the state in the other I/O paths."*

### WDFFILEOBJECT Event Processing Callbacks

The WDF File Object supports the following Event Processing Callbacks provided as part of the **WDF_FILEOBJECT_CONFIG** structure: *EvtDeviceFileCreate*, *EvtFileCleanup*, and *EvtFileClose*. Let's examine the purpose of each of these Event Processing Callbacks in turn.

### *EvtDeviceFileCreate*

While slightly odd in its naming, *EvtDeviceFileCreate* is the Event Processing Callback raised when a WDF File Object is created. This would be, for example, as a result of a user application's call to **CreateFile**. It is during the processing of this routine that the driver instantiates any unique state for this open instance.

The *EvtDeviceFileCreate* function prototype is as follows:

```
VOID
EvtDeviceFileCreate(
   _In_ WDFDEVICE Device,
   _In_ WDFREQUEST Request,
   _In_ WDFFILEOBJECT FileObject
   );
```

Note that we are provided a WDF Device Object handle, which represents the WDF File Object's target device. We are also provided a WDF Request Object handle, which is the WDF abstraction of the native I/O operation representing the creation of the File Object. As part of handling this Event Processing Callback, the driver must call **WdfRequestComplete** on this request to indicate the result of the operation to the requestor. As an aside that we will revisit later, this WDF Request Object is unique in KMDF in that it is not *queue presented*, meaning that it has no parent WDF Queue Object.

Lastly, we are provided the handle of the WDF File Object that is currently being instantiated.

### *EvtFileCleanup*

*EvtFileCleanup* is the Event Processing Callback invoked when the native File Object enters the *cleaned up* state. That is to say, this event is raised when the native handle count drops to zero. This state is triggered directly as a result of the application's call to **CloseHandle**.

The *EvtFileCleanup* function prototype is as follows:

```
VOID
EvtFileCleanup(
  _In_ WDFFILEOBJECT FileObject
  );
```

Unlike in the case of *EvtDeviceFileCreate*, we are **not** provided a WDF Request Object handle as *EvtFileCleanup* operations. This is because Cleanup operations are always assumed to be successful. Note however that cleanup operations do arrive at the Framework as native I/O requests, so KMDF is completing these requests on our behalf "underneath the covers."

### *EvtFileClose*

*EvtFileClose* is the Event Processing Callback raised when the native File Object enters the *closed* state. That is to say, this event is raised when the native reference count drops to zero. This state is triggered some time after the application's call to **CloseHandle.**

(CONTINUED ON PAGE 7)

## WDF File Object Callbacks... (Cont.)

(CONTINUED FROM PAGE 6)

You might notice that this Event Processing Callback seems similar to *EvtFileCleanup*. Both are ultimately called as a result of the application calling **CloseHandle**, so why is there a distinction? In *EvtFileCleanup*, we are simply notified that the application has closed its handle and can no longer use the native File Object to perform I/O. However, there still may be I/O operations submitted via the File Object that have yet to complete. Once the last I/O completes, the File Object is no longer in use and the *EvtFileClose* Event Processing Callback is raised.

In general, this is why it makes sense to defer tearing down any unique state for this open instance until the *EvtFileClose* Event Processing Callback. This eliminates any races between tearing down the open instance state and using the state in the other I/O paths.

The *EvtFileClose* function prototype is as follows:

```
VOID
EvtFileClose(
   _In_ WDFFILEOBJECT FileObject
   );
```

Once again, we are not provided a WDF Request Object handle in this callback, even though close operations also arrive at the Framework as native I/O requests.

### WDFFILEOBJECT Properties

In addition to the Event Processing Callbacks, the **WDF_FILEOBJECT_CONFIG** structure supports the setting of the following properties: *AutoForwardCleanupClose* and *FileObject Class.*

*Property: AutoForwardCleanupClose*
*Type: WDF_TRI_STATE*

In general, create requests are ultimately completed by the Functional Device Object (FDO) owner in the stack. Given that the FDO owner completes the create request, it then makes sense that the FDO owner would be responsible for completing the accompanying native I/O operations for cleanup and close notification.

A filter driver, on the other hand, typically does not play an active role in create, cleanup, or close processing. The filter driver may want to be notified of these events, but would not complete the native I/O operations. Instead, the filter would perform the necessary work and then pass the requests on to the next lower driver.

The *AutoForwardCleanupClose* property tells the Framework which default processing path to take for cleanup, close, and, despite the name, create operations. If the property is set to **WdfFalse**, the FDO based processing is chosen and the I/O requests are completed by the Framework. If **WdfTrue**, the filter driver processing is taken and the requests are passed on. **WdfDefault** chooses the correct behavior based on target device type.

*Property: FileObjectClass*
*Type: WDF_FILEOBJECT_CLASS*

The *FileObjectClass* property is overloaded and used to specify two unrelated options. The first option is whether or not the driver requires WDF File Object at all. By specifying a value of **WdfFileObjectNotRequired**, the Framework will short circuit its processing and raise the *EvtDeviceFileCreate* Event Processing Callback with a NULL *FileObject* parameter.

This may seem like a very odd option and, in fact, it is! A typical driver receiving standard create, cleanup, and close requests would almost never set this option. However, there is a very specific use case that is being addressed here. Namely, the Framework must have a model for handling non-standard, driver-generated create requests that do not contain valid native File Objects. The O/S itself will never send these, but a pair of cooperating drivers may leverage them to establish a connection.

Assuming that a File Object is required, the next series of values indicate optimizations that the Framework may take to facilitate

(CONTINUED ON PAGE 20)

# A Debugger's Delight
## Windows Pool Manager

In our years of experience debugging in the Windows environment, one of the most useful debugging techniques we've found involves the information we can collect from the pool allocator in the kernel environment. Over time, Microsoft has further improved the efficacy of this for debugging by including ever more information with both the debugger tools as well as by increasing the number of data structures described in the public symbols.

In this article, we will provide a basic explanation of how pool management in Windows works from the perspective of someone debugging on the platform. Thus, our goal is not to describe the nuances and details of exactly how the pool manager works, but rather to describe how pool allocation works vis-à-vis what we need to know to more effectively use the debugger.

### Types of Pool Memory

In the Windows kernel environment, we have two basic types of pool: **paged pool** and **non-paged pool**.

Paged pool are blocks of memory where the virtual addresses have meaning, but the mapping from virtual to physical page may not be defined. In a case where the virtual address has no corresponding physical address, the data is stored in a paging file. Like any other virtual address, if a virtual page is accessed for which there is no corresponding physical page, a page fault occurs and the Memory Manager will allocate a new physical page, fetch the data from the paging file into the physical page and then update the page table so that it points to the correct physical page.

Paged pool is typically used for data structures that need not be always resident in memory for correct operation; there is generally more paged pool than non-paged pool and paged pool has less impact on overall machine performance because it does not lock down physical memory. For example, we do quite a bit of file systems work and we routinely work with structures that are only accessed at IRQL < DISPATCH_LEVEL. In such cases, the data structures may be placed in paged pool.

Non-paged pool are blocks of memory where the virtual addresses have meaning and the mapping of virtual to physical page is defined. Thus, if a "non-paged address" is accessed and the virtual-to-physical page is not defined, the Windows Memory Manager will terminate system operation with the bug check 0x50 (PAGE_FAULT_IN_NONPAGED_AREA).

Non-paged pool is typically used for data structures that must be resident in memory for correct operation. For example, any data that is needed to access the paging file (which is where paged pool data may be stored) must be in non-paged pool.

### Windows Pool Allocator

Regardless of the type of pool, Windows uses the same pool allocation logic – it just maintains separate pools with different characteristics.

Dynamic memory allocation is an essential part of most kernel components as it provides a general framework for allocating resources on demand and thus permitting considerable flexibility in resource utilization. Indeed, the trend in Windows has been to encourage the use of dynamically allocated structures and discourage the use of statically allocated structures, as the former leads to greater flexibility in terms of implementation.

The core Windows OS provides two key functions for utilizing its dynamic memory allocator:

- ExAllocatePoolWithTag – this function allocates blocks of memory for use by a kernel component, including a driver.
- ExFreePool – this function frees a previously allocated block of memory to a kernel component.

The Windows pool allocator splits allocation requests into two categories – those that are small enough to use the small pool allocator (some value less than **PAGE_SIZE** so that it makes sense to fit multiple entries on a single page) and the large pool allocator, which allocates blocks of memory in integral multiples of **PAGE_SIZE**.

For large pool allocations, the actual allocation tracking information is stored separately from the block of memory and thus is handled completely differently. Most blocks of pool memory allocated by drivers are typically much smaller than a single page.

For allocations smaller than one page, the small pool allocator is used to satisfy the request. For allocations of one page or more they are allocated by the large pool allocator – in multiples of **PAGE_SIZE**.

(CONTINUED ON PAGE 9)

## Windows Pool Manager (Cont.)

(CONTINUED FROM PAGE 8)

### Pool Block (Small Pool Allocations)

Memory allocated by the small pool allocator consists of a header, data region and special computed value (“canary”). **See Figure 1**.

Note that the “canary” value follows the data region and provides a mechanism for detecting buffer overwrite error. While special pool can also be used to detect buffer overruns, the canary technique is always active (beginning with Windows Server 2003) and does not require consumption of two pages of virtual address space to implement.

*Aside: it is called a canary as an early warning sign of an imminent problem. Coal miners would use canaries as a form of “early warning sign” in coal mines, so when the canary stopped singing the miners knew something was wrong. http://www.wisegeek.org/what-does-it-mean-to-be-a-canary-in-a-coal-mine.htm*

When a canary overwrite is detected, Windows will raise a bug check. This applies to all drivers in the system, not just those drivers that are running under Driver Verifier.

The first portion of any block of pool memory is the _POOL_HEADER structure, which defines the current layout of the header used by the small pool allocator. Here is a current version of it from a Windows 8 x64 box:

```
0: kd> dt nt!_POOL_HEADER
   +0x000 PreviousSize     : Pos 0, 8 Bits
   +0x000 PoolIndex        : Pos 8, 8 Bits
   +0x000 BlockSize        : Pos 16, 8 Bits
   +0x000 PoolType         : Pos 24, 8 Bits
   +0x000 Ulong1           : Uint4B
   +0x004 PoolTag          : Uint4B
   +0x008 ProcessBilled    : Ptr64 _EPROCESS
   +0x008 AllocatorBackTraceIndex : Uint2B
   +0x00a PoolTagHash      : Uint2B
```

This pool header is used to keep track of the state of the each individual block of pool memory within a given page.

### *Debugger “!pool” Command*

The debugger has a special extension command for looking at a given memory location to see if it exists within a block of pool. This command – the **!pool** – command – will examine the entire page of memory, looking to identify the pool block that contains the given address.

If the address given is not a valid pool address, it will typically display some sort of error condition, since the memory will not be laid out in the proper format.

```
0: kd> !pool fffffa8016990040
Pool page fffffa8016990040 region is Nonpaged pool
*fffffa8016990000 size:  540 previous size:    0  (Allocated) *Thre
                   Pooltag Thre : Thread objects, Binary : nt!ps
 fffffa8016990540 size:   30 previous size:  540  (Allocated)  AlSe
 fffffa8016990570 size:  210 previous size:   30  (Allocated)  ALPC
 fffffa8016990780 size:  880 previous size:  210  (Allocated)  LSfR
```

Note that the allocated block in which the address given to **!pool** was given is indicated by the ***** at the beginning of the line.

The debugger extension exploits its knowledge of the pool block layout to analyze the page. Each header contains information that is sufficient to “walk” the list of pool allocations for that entire page, with each byte on the page either being part of the pool manager meta-data (header or canary) or the data region as well as some padding.

Note that the Windows small pool allocator guarantees the memory returned will be on an eight byte boundary. There are parts of Windows that still exploit this knowledge and sometimes they use the low two bits of an address for storing information – but those bits are not really part of the memory address.

When a driver allocates a block of pool, the pool allocator will attempt to find an existing block of pool of the correct size. This is done by maintaining lists of free pool regions. Typically, this has been implemented by using the data region as a doubly-linked list pointer so that the memory can be stored on a list of the relevant size – though we note that nothing requires this implementation and the details of the pool allocator do change from release to release.

(CONTINUED ON PAGE 22)

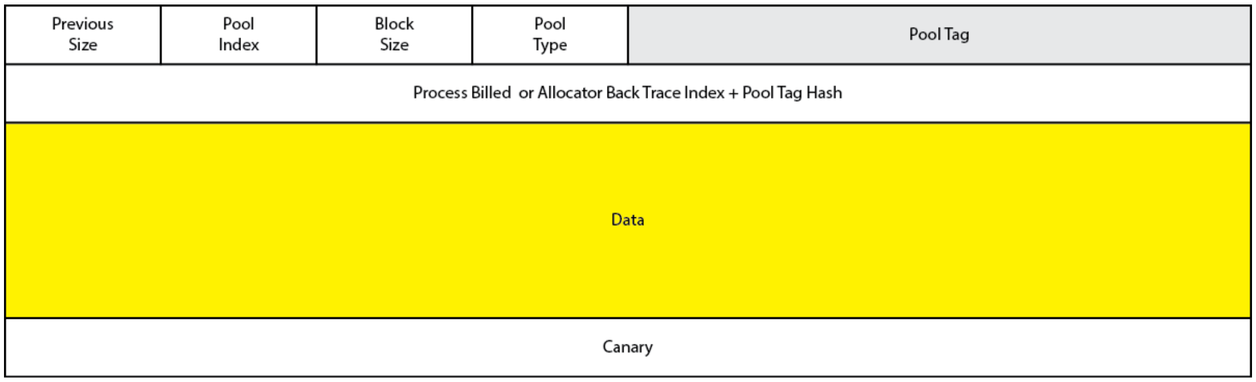
Previous Size
Pool Index
Block Size
Pool Type
Pool Tag
Process Billed or Allocator Back Trace Index + Pool Tag Hash
Data
Canary

**Figure 1— Small Pool Allocations**

# Analyst's Perspective
# The Truth Behind PAGE_FAULT_IN_NONPAGED_AREA

In our debugging seminars, we make a big show of telling our students to not only run **!analyze –v** when looking at a system crash, but to also actually *read the output* of the command. Note that I didn't say to skim it casually. Nor did I say to glaze over while the output flies by (*Hmmm…I wonder if they put more snacks out?*). But actually *read* and *understand* what the bugcheck analysis says. Ultimately, the bugcheck code and description are the reason for the system crash, so if you don't start out with a complete understanding of the **!analyze –v** output then you're hopeless from the start.

It's my firm belief in this step that makes me want to claw the text off the screen when I see the description accompanying the all too common PAGE_FAULT_IN_NONPAGED_AREA bugcheck:

```
0: kd> !analyze -v

*****************************************************
*                                                   *
*                Bugcheck Analysis                  *
*                                                   *
*****************************************************

PAGE_FAULT_IN_NONPAGED_AREA (50)
Invalid system memory was referenced.  This cannot be
protected by try-except, it must be protected by a Probe.
Typically the address is just plain bad or it is pointing
at freed memory.
```

In order to understand what my problem is with this description, let's break it down into individual pieces:

*`Invalid system memory was referenced.`*

This is absolutely correct, no problem here. The PAGE_FAULT_IN_NONPAGED_AREA bugcheck only ever occurs when you dereference an invalid kernel address. The Windows Page Fault Handler has to assume that if an invalid kernel address has been dereferenced something seriously bad is going on, so it has no other choice than to crash the machine.

*`This cannot be protected by try-except,`*

Agreed! As I mentioned, the Page Fault Handler simply bugchecks if an invalid kernel address is dereferenced. This is different than if an invalid *user* address is dereferenced. In that case, the Page Fault Handler raises an exception that can be caught with a __try/__except block. This allows the O/S and drivers to be resilient to malicious or poorly written applications that supply invalid user buffers to I/O operations.

*`it must be protected by a Probe.`*

And so it begins…This statement is a vague half-truth, which makes it all the more annoying. If it was all wrong I could simply tell you to ignore it. If it was all correct I wouldn't have to bother writing this article and I could be outside making snow angels.

For starters, the *Probe* that the sentence is referring to is in fact two different functions: **ProbeForRead** and **ProbeForWrite**. This would lead you to believe that you could avoid dereferencing invalid kernel memory if you called one of these functions before dereferencing the pointer, right? Sort of! From the docs:

*The **ProbeForRead** routine checks that a <u>user-mode buffer actually resides in the user portion of the address space</u>, and is correctly aligned.*

And:

*The **ProbeForWrite** routine checks that a <u>user-mode buffer actually resides in the user-mode portion of the address space</u>, is writable, and is correctly aligned.*

All these APIs really do is make sure that a buffer pointer is a user mode pointer. If the pointer is a kernel mode pointer, they raise an exception that the caller can catch. Calling these APIs before dereferencing any pointer supplied by user mode is a required step in buffer validation, otherwise you run the risk of dereferencing a kernel mode address provided by a user mode caller and generating a PAGE_FAULT_IN_NONPAGED_AREA bugcheck.

In any other context these APIs make no sense for a driver. If you protected all of your kernel mode references with a Probe you would never actually dereference a kernel mode pointer. This would surely prevent you from dereferencing invalid kernel memory, so I suppose the statement isn't *entirely* inaccurate, but your driver wouldn't be very useful.

*`Typically the address is just plain bad or it is pointing at freed memory.`*

The situation has returned to normal. The bugcheck is ultimately caused by an invalid kernel pointer, so you have all of the usual reasons to look out for when it comes to why the pointer might be bad. Sure, it could be that you didn't call a Probe function on a pointer from a buggy or malicious application, but in general this guidance will point you in the right direction.

*Analyst's Perspective is a column by OSR Consulting Associate, Scott Noone. When he's not root-causing complex kernel issues, he's leading the development and instruction of OSR's Kernel Debugging Seminar. Comments or suggestions for this or future Analyst Perspective columns can be addressed to ap@osr.com*

Follow us!

## Guest Article
## Driver Installers

**By Tim Roberts, community contributor**

Installers. The mere word can strike an icy cold stab of fear into the hearts of driver developers. It's not clear why that should be the case; after all, installers run in the relatively safe comfort of user mode, where the APIs are well developed, there are few sharp edges, and the penalty for mistakes is smaller. However, I understand the feeling. I avoided installers as if they were infectious, until I finally decided to dig in and see what the fuss was about.

In this article, I'm going to try to alleviate any fears you may have about driver installers, and describe for you a relatively simple but complete mechanism for installing drivers. One of the problems with installers is that there are a lot of options. I'm not claiming my method is the best one, but it is one that has worked for me for many years.

### Which Driver Type?

There are a number of different types of drivers, and those driver types require different types of installers. A partial list might include:

1. PnP drivers
2. Device filter drivers
3. Class filter drivers
4. Legacy drivers

PnP drivers require an INF file. That's the most common type of driver (I assert without evidence), so that's the type I'm going to focus on in this article. I will make some comments about class filter drivers and legacy drivers towards the end.

There are several distinct steps required in order to get a driver package installed and operational. In this article, I'm going to divide the process into three steps. To make things more interesting, I will discuss those steps in reverse order.

### Installation

The final part of the driver installation process is that part we usually think of as "installation". It is the process of copying files into their operational locations (\Windows\Inf, \Windows\System32, and \Windows\System32\Drivers), creating services, making registry entries, loading the driver, and calling the driver's initial entry points.

For PnP drivers, this part of the installation process is handled entirely by Device Manager, as directed by the INF file. When some bus driver reports the creation of a PDO to the PnP Manager, and that PDO has a device type that is not already known to the system, Device Manager asks for its hardware ID, and goes on a treasure hunt to find a driver to handle it. It first looks through all of the INF files that have previously been installed in the \Windows\Inf directory. If there is no match there, it searches through the driver store, which contains the pre-installed driver packages.

When Device Manager finds a matching driver package, it "executes" the INF file, copying files into the System32

(CONTINUED ON PAGE 12)

## Driver Installers (Cont.)

(CONTINUED FROM PAGE 11)

directory, creating services, registering DLLs, making registry entries, running coinstallers, then loading the driver into memory and executing it.

In order for Device Manager to do its job, the driver package has to be pre-installed into the driver store for the PnP Manager to find. That is the next stop in our backwards tour of the installation process.

### Pre-Installation

Pre-installation is the process of copying a driver package to the "driver store". On Windows XP, the driver store is in \Windows\System32\Drvstore. On Vista and later systems, the driver store is in \Windows\System32\DriverStore.

The key API in this process is **SetupCopyOEMInf**. You can call **SetupCopyOEMInf** from your own application, or you can use "devcon dp_add". However, given that you can't ship the "devcon" utility with your product, the most convenient option is to use the DPInst tool that is included in the WDK in redist\difx\DPInst. You are allowed to redistribute the DPInst executables in your own installer packages.

DPInst is a narrowly focused application. In its simplest use case, it displays a welcome dialog, then calls **SetupCopyOEMInf** on every INF file it finds in the same directory as the executable. It then calls **UpdateDriverForPlugAndPlayDevices**, which forces Device Manager to take another look around for devices that need drivers. In some cases, this is exactly what you need. If you need to do something a bit different, DPInst's activities can be customized through the use of an XML file. For example, you can have DPInst display an end-user license agreement, you can customize the welcome and finish messages, you can customize the icon, and you can add a bitmap to the wizard dialog it displays. DPInst even registers an uninstaller for you, listed in Control Panel as "Windows Driver Package" with your hardware's description. It's important to note that you must use the 64-bit DPInst on a 64-bit system and a 32-bit DPInst on a 32-bit system.

The DPInst.xml file needs to be in the same directory as the DPInst executable. See the example in **Figure 1**.

At the bottom, after the <language> block, you'll see a list of standalone tags selecting various DPInst options. All of these options have equivalent command-line switches, if you prefer to go that route. The <legacyMode/> switch is important; without it, DPInst will not handle non-WHQL-signed packages. The <deleteBinaries/> options tells the DPInst uninstaller to remove any binaries it created when it installed. The <suppressAddRemovePrograms/> switch tells DPInst not to create an uninstaller at all. I do that only because I have my own installer handle that step (which you will see shortly), and it confused users to have two entries.

For DPInst to do its job, it needs to be in the same directory as the root of your driver package (along with DPInst.xml). For testing, it's easy enough to copy your driver package onto a test system and run DPinst by hand. You might even be able to gather everything up into a zip file and distribute that to members of your own team. However, it should be clear that what has been described here does not provide the installation experience that most end users want. Users expect to have a single executable that does the whole job. For that, we move back in time one more step.

### Pre-Pre-Installation

Pre-pre-installation is what I call the step of getting your driver package (along with DPInst) loaded on to a client's computer, and then running DPInst to do a pre-installation. In the spirit of full disclosure, you need to know that the term "pre-pre-installation" is one that I made up for this article. Don't go searching for it.

(CONTINUED ON PAGE 13)

```
<xml version="1.0" >
 <dpinst>
    <!-- English -->
    <language code="0x0409">
     <dpinstTitle>XYZ Sonic Screwdriver Driver Installation</dpinstTitle>
     <welcomeTitle>Welcome</welcomeTitle>
     <welcomeIntro>Welcome to the XYZ Sonic Screwdriver Driver Setup program. This program will install the XYZ Sonic Screwdriver
         Drivers on your computer.</welcomeIntro>
     <installHeaderTitle>Please wait while Setup finishes installing the driver.</installHeaderTitle>
     <finishTitle>Setup complete</finishTitle>
     <finishText>Setup has finished installing the Sonic Screwdriver Drivers.</finishText>
    </language>
    <legacyMode/>
    <forceIfDriverIsNotBetter/>
    <deleteBinaries/>
    <suppressEulaPage/>
    <enableNotListedLanguages/>
    <suppressAddRemovePrograms/>
 </dpinst>
```

Click to Expand

Figure 1

## Driver Installers (Cont.)

(CONTINUED FROM PAGE 12)

The pre-pre-installation process is just a normal application-style installation, and can be done using traditional installer tools. Many people use WiX, which is an XML-based utility that builds Microsoft Installer (MSI) files. Some people use InstallShield. I happen to use NSIS, the Nullsoft Scriptable Installer System, at http://nsis.sourceforge.net. It is an open source command-line tool that compiles a script into an executable.

Whichever tool you use, your pre-pre-installer has two basic jobs: copy the driver package onto disk, and run DPInst. In this section, I will show you a simple NSIS script to preload and pre-install a driver package.

One of the things that makes the pre-pre-install script tricky is the 32-bit/64-bit problem. You need to decide whether you want two separate installers (one for each bittedness), or one installer with both drivers. For mostly historical reasons, I tend to build my driver packages with one INF and two subdirectories. However, the samples in the WDK have all gone the other direction, where you get one driver package, complete with INF, per architecture. The script I'm about to demonstrate assumes this layout.

The NSIS script language is an odd beast, somewhere between batch files and the Basic language. It has some rather primitive constructs that derive from being a single-pass compilation process. I'll point those out when they come up.

See **Figure 2** (P. 14) for a complete NSIS script to create a moderately-featured installer for fictional "Sonic Screwdriver" from the XYZ Company.

The script begins with a set of global declarations, followed by a set of sections, each of which has instructions for a particular part of the installer. It is possible to create a script with multiple selections that presents a menu to the user, allowing them to choose which subcomponents to include. I have not done that here.

The initial section declares the installer name, the dialog caption, and the default install directory name (in this case, \Program Files\XYZ\SonicScrewdriver). $PROGRAMFILES is an NSIS variable that expands to the proper Program Files directory. (Because NSIS creates a 32-bit executable, this actually expands to "Program Files (x86)" on a 64-bit system.) The OutFile statement then defines the name of the executable that NSIS will create.

After that, we start the main installation section. The first thing I do here is check the registry to see if a previous run of this installer registered an uninstaller. If it did, I run the uninstaller to clear out any previous instances of the driver package. End-users expect an installer to be able to upgrade itself in place; I discovered that repeated instructions to manually run the uninstaller first were futile.

After that, we begin the process of copying files. The SetOutPath statement tells the installer where the files should go. We could have additional SetOutPath statements to create a directory tree. For example, when I have a driver package that includes both 32-bit and 64-bit drivers in a single package, I use statements similar to the following to place the files accordingly:

(CONTINUED ON PAGE 14)

## Driver Installers (Cont.)

(CONTINUED FROM PAGE 13)

```
SetOutPath $INSTDIR\32
File “drv\32\driver.sys”
File “drv\32\helper.dll”
… other 32-bit files …

SetOutPath $INSTDIR\64
File “drv\64\driver.sys”
File “drv\64\helper.dll”
… other 64-bit files …
```

One of the tricky things to keep track of is whether a particular directive applies to the build machine or the eventual client machine. For this particular example, I'm assuming we have separate driver packages for 32-bit and 64-bit. In that case, I only need to copy one package onto any given client. The IfFileExists statement checks to see whether a SysWOW64 directory exists; if it does, then this is a 64-bit machine, and so I load the 64-bit driver package. The conditional directives in NSIS are not modern structured programming constructs. The way ”if” works in NSIS is if the condition is true (in this case, if $WINDOWS\SysWOW64\* exists), control jumps to the statement given by the 3rd parameter. If the condition is false, control jumps to the statement given by the 4th parameter. The statement can be specified as a signed integer (+3 meaning “skip to the 3rd line following this one), or with a named label. In this case, the 0 means to fall through to the next line if t he directory exists, otherwise jump to the label else1.

Each File directive identifies a file to be copied into the current SetOutPath directory. The directive specifies the file's location on the disk where the compilation is being done. The file will keep the same name, although you can specify a new name if you wish.

After copying the files for the driver package and the appropriate WDF co-installer, I add the DPInst.xml file to the package. That completes the file list.

The next step is to run DPInst itself (on the client system). This is accomplished by the ExecWait statement. As the name implies, the script will not continue until the DPInst command completes.

After that, all that's left is to create and register an uninstaller. We specify the uninstaller's name with the WriteUninstaller directive, and we specify the actions it has to take in the separate “Uninstall” section. All we have to do is run DPInst with the “/u” parameter, to have it undo all the magic it did, and then delete any files and directories we created, and any registry keys we created.

(CONTINUED ON PAGE 15)

```
Name "XYZ Sonic Screwdriver Driver"
Caption "XYZ Sonic Screwdriver Driver"
InstallDir "$PROGRAMFILES\XYZ\SonicScrewdriver"
DirText "This will install the XYZ Sonic Screwdriver drivers on your
    computer.  You may choose a directory to hold the driver files:"
OutFile "setup_ssd.exe"

; The stuff to install
Section "Driver"

 ; If the uninstall key exists, run DPInst to uninstall.
 ReadRegStr $0 HKLM Software\Microsoft\Windows\CurrentVersion
                   \Uninstall\xyzsonic" \ "UninstallString"
 StrCmp $0 "" after
    ; We always put double-quotes in path, so $0 starts with one.
    StrCpy $1 $0
    loop:
     IntOp $1 $1 - 1
     StrCpy $2 $0 1 $1
     StrCmp $2 "\" 0 loop
    StrCpy $0 $0 $1
    StrCpy $0 '$0\DPInst.exe" /q /u $0\xyzsonic.inf"'
    DetailPrint "Uninstalling old driver"
    DetailPrint $0
    ExecWait $0
after:

 ; Copy the files.
 SetShellVarContext all
 SetOutPath $INSTDIR

 IfFileExists $WINDIR\SysWOW64\*.* 0 else1
    File "\ddk\7600\redist\DIFx\DPInst\EngMui\amd64\DPInst.exe"
    File "..\Lag\amd64\xyzsonic.sys"
    File "..\Lag\amd64\xyzsonic.inf"
    File "..\Lag\amd64\xyzsonic.cat"
    File "..\Lag\amd64\WdfCoInstaller01009.dll"
      Goto endif1
 else1:
    File "\ddk\7600\redist\DIFx\DPInst\EngMui\x86\DPInst.exe"
    File "..\Lag\x86\xyzsonic.sys"
    File "..\Lag\x86\xyzsonic.inf"
    File "..\Lag\x86\xyzsonic.cat"
    File "..\Lag\x86\WdfCoInstaller01009.dll"
 endif1:

 File "DPInst.xml"

 ; Install the driver.
 ExecWait '"$INSTDIR\DPInst.exe" /lm'

 ; Create the uninstaller.
 WriteRegStr HKLM
"Software\Microsoft\Windows\CurrentVersion\Uninstall\xyzsonic" \
             "DisplayName" "XYZ Sonic Screwdriver"
 WriteRegStr HKLM
"Software\Microsoft\Windows\CurrentVersion\Uninstall\xyzsonic" \
             "UninstallString" '"$INSTDIR\Uninstall.exe"'

 WriteUninstaller "Uninstall.exe"

SectionEnd ; end the section

Section "Uninstall"
 SetShellVarContext all
 DeleteRegKey HKLM
"Software\Microsoft\Windows\CurrentVersion\Uninstall\xyzsonic"
 ExecWait '"$INSTDIR\DPInst.exe" /u xyzsonic.inf'

 Delete $INSTDIR\DPInst.exe
 Delete $INSTDIR\DPInst.xml
 Delete $INSTDIR\xyzsonic.sys
 Delete $INSTDIR\xyzsonic.inf
 Delete $INSTDIR\xyzsonic.cat
 Delete $INSTDIR\WdfCoInstaller01009.dll
 Delete $INSTDIR\Uninstall.exe
 RMDir $INSTDIR
SectionEnd
```

Figure 2

## Driver Installers (Cont.)

(CONTINUED FROM PAGE 14)

This script can be extended in an infinite number of ways. The NSIS package provides a large number of samples and plugins to perform additional functions, most of which don't really apply in the driver world. It is possible to pass parameters to the "makensis" command line, similar to pre-processor variables on the C compiler command line. For example, I have one script that can build either a "checked" or a "free" installer, based on the contents of a command-line parameter.

### Other Driver Types

As I noted at the beginning, the above description applies to PnP drivers. Other types of drivers have different requirements. Legacy drivers and class filter drivers, for example, can both be installed using an INF with a [DefaultInstall] section. In that case, you can use a very similar NSIS script to copy a driver package without DPInst, and then execute RunDll32 to run that section:

```
 rundll32 setupapi.dll,InstallHinf DefaultInstall 132
c:\install\mydrv.inf
```

However, this mechanism has the huge disadvantage that it does not run the WDF co-installer. This is a non-PnP installation, and co-installers only run for a PnP installation. This problem was described in *The NT Insider* back in March of 2008 (see the article http://www.osronline.com/article.cfm?article=446). If your driver package needs to run on a system that might have an older version of KMDF than the one you need, you will have to supply an application. Device filter drivers also generally require a custom application, because it is necessary to identify the exact hardware IDs to be filtered. Even in these cases, however, one could have an NSIS script copy the driver file plus the installer application, and then run the installer.

### Conclusion

I hope this article has alleviated some of the horror you might have felt towards driver installations. Like most computer tasks, installation becomes much more manageable when you chop it up into smaller pieces and tackle those pieces.

*Tim Roberts has been wrangling computers from mainframes to micros for 40 years, and has been part of the Windows driver world since Windows 3.0. An 18-year MVP, Tim is a principal in P&B, a consulting company providing custom hardware and software solutions to difficult computing situations. Tim can be reached at timr@probo.com.*

## Getting Started Writing Windows Drivers (Cont.)

(CONTINUED FROM PAGE 4)

mystic project files and compile and link your code from the command line. Today, Windows driver development is fully integrated with Visual Studio.

At the current time (January 2014, Windows 8.1 was released a few months ago), driver development is supported in Visual Studio 2012 and Visual Studio 2013, Professional Edition or better. That means the "free for anybody to use" version of Visual Studio (Visual Studio Express Edition, or whatever they're calling it now) cannot be used for driver development. So you will need to buy an MSDN and Visual Studio subscription. One very cool thing to note, however, is that you **can** use the 90-day trial versions of Visual Studio Professional 2013 and Visual Studio Ultimate 2013 that are available and free to download.

Once you have Visual Studio purchased and installed on your development machine, you'll also need to install the Windows Driver Kit (WDK) add-in that supports driver development. This is a separate, but free (yay!), download from Microsoft (no MSDN subscription necessary). Search "Get Windows Driver Kit" using your search engine of choice.

Visual Studio and the WDK together provide everything you need to create driver projects, and to compile, link, and even debug Windows drivers. After you've successfully installed Visual Studio and the WDK, you can very easily build a simple driver demo project. You don't even need any hardware! Just select "New Project" and within *Visual C++* select the *Windows Driver* project category. Within this category select *Kernel Mode Driver (KMDF)*. Click OK and Visual Studio will generate a simple starter or demo driver project for you that doesn't require any specific hardware. This driver will successfully build, and can even be installed on a test machine. Yup, it really **is** that simple.

Ah, test machines. That's probably something we should discuss. Driver development on Windows requires two Windows systems. One system where you run Visual Studio, do your development, and run the debugger. And a second, separate, system on which you run your driver. The Windows kernel debugger, running on your Development System, controls your Target System (where the driver you're developing is running) via a remote connection that can be either a serial port, 1394, the network, or even in some cases USB. **See Figure 1** (P. 17).

If you think about it, this makes good sense: Driver and hardware errors can quite easily destabilize or even crash a system. So you certainly don't want to be running your new and potentially buggy driver on the same system on which you're editing your source code files and doing your development.

In many cases, the second system can be a virtual machine. Using a virtual machine is acceptable when you're writing a

(CONTINUED ON PAGE 17)

## Understanding OS Concepts and Windows Concepts

### Windows Internals 6th Edition -- Part 1
### (Russinovich, Solomon, Ionescu) (Microsoft Press)

This is **the** basic description of Windows OS Architecture. Everyone in the world of Windows has read it at some time. When you read the following chapters, you may just skip the exercises shown or try a few if they sound interesting to you… it's your choice.

- Chapter 1: Concepts and Tools (whole chapter)
- Chapter 2: System Architecture (whole chapter)
- Chapter 3: System Mechanisms (Up to but not including section entitled *Advanced Local Procedure Calls(ALPC))*

### Windows System Programming 4th Edition
### (Johnson M. Hart) (Addison-Wesley Microsoft Technology Series)

If you're going to write device drivers, it probably makes sense to understand something about how to write Windows programs. If you've worked on Unix, and you've never written a program on a Windows system, this book will give you a lot of the information you'll need.

- Chapter 1: Getting Started With Windows (whole chapter)
- Chapter 2: Using the Windows File System and Character I/O (whole chapter)
- Chapter 4: Exception Handling (whole chapter)
- Chapter 14: Asynchronous Input/Output and Completion Ports

## Getting Started Writing Windows Drivers (Cont.)

(CONTINUED FROM PAGE 16)

driver (such as a filter driver or a file system) that doesn't directly access any hardware. But if your driver talks to real hardware, you'll need a real, physical, second machine to use as your Target System. This is true even when you're building a driver for something like a USB device, when the VM host you're using allows you to assign access to the device exclusively to a given VM.

We mentioned the Windows kernel debugger. This debugger is named WinDbg (which almost everyone pronounces as "wind bag", by the way). The debugger is included in the Windows Driver Kit and is automatically installed on your system when you install the WDK. It's the debugger you'll use as part of developing and testing your driver. It's very similar to the user-mode debugger in Visual Studio, and has most of the same features.

There are several options available for using WinDbg for debugging your driver. One option is to use WinDbg directly within Visual Studio, through the interface provided by the WDK. While this pretty much works, here at OSR we don't recommend this. Our experience is that trying to use WinDbg from within Visual Studio creates more complications than it's currently worth. Instead, we recommend that you run WinDbg directly from your development machine, outside of Visual Studio. This allows you to use Visual Studio for driver development, which is what it's best at, and use WinDbg directly for debugging, which is what WinDbg is best at.

Before you can use WinDbg to debug your driver, you'll need to enable kernel debugging on the target system. Fortunately, it's easy and very well documented (thank you, WDK doc writers). Search "Setting Up Kernel-Mode Debugging Manually" in your search engine of choice for the steps.

One quick note about debugging. Do not, under any circumstances, try to develop your driver without setting up WinDbg. For some reason, there are folks who've been fooled into thinking they can use something like the Microsoft DebugView utility, which allows DbgPrint statements (the kernel -mode equivalent of printf or OutputDebugString) from your driver to be viewed on your system, as their sole tool for driver development. While DebugView can be useful at times, we can guarantee that it is no substitute for having a debugger that allows you to set breakpoints, single step, and change the contents of structure fields and local variables. While setting up WinDbg for the first time can sometimes be annoying, we promise you it'll be worth your effort in the long run. Yay, WinDbg!

The final thing you'll need are the Windows Driver Kit Samples. These are example drivers, provided by Microsoft, that demonstrate how to write drivers of various kinds. They're just like the typical sample code you download from MSDN: They are very useful and highly instructive, even if some of the code provided isn't always exactly "the best." Samples are provided for all sorts of hardware drivers, filter drivers, and software-only drivers. Heck, they even give you the source code to a few of the drivers that are part of the Windows OS… including sources for the FAT file system.

The samples are available as a separate download from Microsoft, and as with the WDK no MSDN subscription is

(CONTINUED ON PAGE 18)

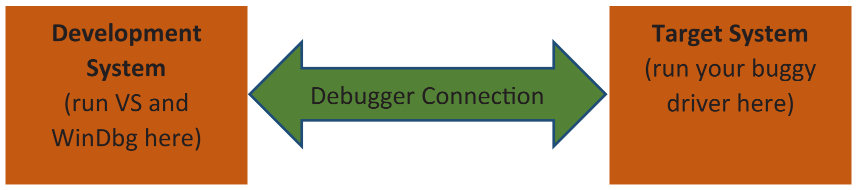

**Figure 1— Basic Windows Driver Development Environment**

DESIGN AND CODE REVIEWS

When You Can't Afford Not To

Have a great product design, but looking for validation before bringing it to your board of directors? Or perhaps you're in the late stages of development of your driver and are looking to have an expert pour over the code to ensure stability and robustness before release to your client base. Consider what a team of internals, device driver and file system experts can do for you.

Contact OSR Sales — sales@osr.com

## Getting Started Writing Windows Drivers (Cont.)

(CONTINUED FROM PAGE 17)

required. Search "Windows hardware development samples" from your search engine of choice. You can download specific samples individually, or you can download the entire ZIP archive (about 100MB when this was written, including more than 160 sample drivers). We recommend you download the complete archive. Take your time and look through the samples. This will be time well spent.

So… now you have the background info you need, and you have all the stuff you need to develop Windows drivers. What's the next step?

### What Driver Model to Use

The actual development of a Windows driver starts with choosing what "driver model" to use for your driver's implementation. Many folks find this step confusing. A driver model is an overall driver organization, including a set of APIs and entry points, which you'll use when you write your code. Unlike some other operating systems that support a small number of driver models ("block" and "character", for example) Windows has a wide number of driver models. The best driver model to choose is based on as many as three things. These are:

- The type of driver you're writing: Hardware device, filter, or some other kind.
- If you're writing a driver for a hardware device, the category (storage controller, sound card, graphics adapter, network card) of device.
- Developer preference

Now hear this: The choice of a driver model is **the most important decision** you'll make about how your driver will be developed. And it's a place where many people make the wrong decision and "go off the rails" – making their project much harder than it needs to be. So take some time to make this decision. Don't simply Google around and find some trash example lying on a web site somewhere and start to hack it. Make the decision thoughtfully.

### General Purpose Models

Broadly speaking, there are two Windows driver models that apply for general use, and some Windows driver models that apply to specific devices. For example, if you're writing a driver for a local area network card, Windows has a specific model that is tailored specifically for this use and makes it maximally convenient to implement this type of driver. Likewise, if you're writing a driver that supports streaming audio or streaming video, Windows has a specific model for these types of drivers. These are only two simple examples. Windows has specific models for lots of other device types as well.

Lacking a specific model for your device type, you can use one of the general-purpose models. The first general-purpose model is the Windows Driver Model (WDM). WDM is the old, historic, model for writing Windows drivers. Nobody should use this model anymore for writing new Windows drivers. Seriously. Nobody. It's hard to use and filled with "traps" that have evolved over years to support backward compatibility guarantees. Trying to write a new WDM driver in the 21st Century will do nothing but make you hate life. Don't do it. Enough said?

Much preferred over WDM is the Windows Driver Foundation (WDF). This is the second general-purpose driver model that Windows supports. WDF is a modern, pleasant, and (dare I say it) almost easy to use method for writing Windows drivers. Unless there's a specific model that Microsoft recommends for the device, filter, or software-only driver you need to write, you'll want to use WDF.

One interesting thing about WDF is that it actually comes in three flavors, called Frameworks:

- Kernel Mode Driver Framework – KMDF
- User Mode Driver Framework V1 – UMDF V1.x
- User Mode Driver Framework V2 – UMDF V2.0 (only applies to Windows 8.1 and later)

KMDF is the Kernel Mode Driver Framework. This is the model you'll almost certainly want to use now and in the near future for any general-purpose Windows driver development project.

You'll notice that there are two WDF Frameworks that allow you to write drivers in user-mode. Writing drivers in user mode is good, because if there's a bug in your driver (let's say, you deference a null pointer) your user-mode driver won't crash the system the way it would if you wrote your driver in kernel mode. That's certainly a **very** good thing, and contributes to nothing but customer satisfactions. So, why didn't we recommend using UMDF for writing your drivers?

Using UMDF today is a problem. UMDF V1 is the older model. It'll support devices running on Windows versions as old as Windows XP. But UMDF V1 uses an odd, difficult, programming pattern that's based on COM (yes, the Component Object Model… **that** COM). Add to that the fact that UMDF V1 has more or less been put in "end of life" status by Microsoft, and you get a model that most people will want to avoid.

UMDF V2.0 is actually a terrific driver model. It uses 99% the same syntax as KMDF, but it runs in user mode, thus contributing to overall system stability. So why don't we recommend using UMDF V2.0 today? Because UMDF V2.0 is currently only supported on Windows 8.1 or later. To be absolutely clear, this means that if you write a UMDF V2 driver, that driver can only be installed on systems that are running Windows 8.1 or more recent versions of Windows. In short,

(CONTINUED ON PAGE 19)

## Getting Started Writing Windows Drivers (Cont.)

(CONTINUED FROM PAGE 18)

unless you only need to support Windows 8.1 or more recent systems, UMDF V2 isn't really a viable choice. On the other hand, if you **do** only need to support Windows 8.1 or later (I don't know, maybe you're writing a driver for some sort of embedded system) then UMDF V2.0 could be a very good choice indeed.

### Choosing the Best Model for Your Project

Confused? It wouldn't be surprising if you are. We told you many people find this driver model stuff confusing. Fortunately, there are some simple rules that can help you decide the best driver model for your use. Here are those rules:

- **Writing a driver for a hardware device?** Check the Windows Hardware Certification Requirements for the type of device that you'll be supporting. To do this, search for "Windows Hardware Certification Requirements: Devices". If the type of device you'll be supporting is listed, the Certification Requirements document will almost always specify the driver model you must use.

  Note that this guidance applies even if you don't plan to apply for Windows Hardware Certification for your device and driver. The Certification Requirements will almost always point you in the direction of the best, easiest, most modern, and most supportable driver model that applies to your type of device.

- **Writing a filter driver?** A filter driver in Windows is a type of driver that monitors I/O operations going to a given device/driver in the system and intercepts those I/O operations. The purpose for intercepting those I/O operations might be to track them, measure them, or modify them. If you're writing a filter for file systems (like for an antivirus product) or networks (such as you would write for a firewall product), there are specific driver models defined for these uses.

- **Writing a software-only driver?** For example, maybe you need to write a driver that collects data in kernel-mode. In this case, you probably want to write a software-only KMDF driver. Using what's called the "legacy NT model" is also a good option. But from the viewpoints of your general knowledge and ease of support, KMDF is probably going to be the right choice.

- **Are you writing a file system?** Stop reading now. You almost certainly do not want to write a Windows file system. It's really difficult. We know, because it's one of the things we've done over the years here at OSR. Send us email. We'll see if we can talk you out of it, and if not we'll point you in the right direction. Seriously. No charge.

- **Neither of the previous steps pointed you to a specific model. Do you need to support systems older than Windows 8.1?** If you only need to support Windows 8.1 or later, the best model for you to use is probably UMDF 2.0.

  If you need to support systems older than Windows 8.1, then your best choice of driver model is probably KMDF.

There are a number of factors that contribute to the decision of which driver model is best for you. You can read more about this on MSDN. Search for the page titled "Choosing a driver model". For the reasons we described above, we recommend for the present time you ignore Microsoft's advice about preferring UMDF. UMDF V2.0 will be a great choice when it supports the majority of systems in the field (either because Microsoft decides to support UMDF 2 on systems older than Windows 8.1 or everyone is running Windows 8.1 or later). But until that time, everywhere you see UMDF recommended we suggest you choose KMDF instead.

### In Summary

That's how you get started writing Windows drivers. Learn a bit about Windows architecture, get the tools, and choose a model for your driver.

Of course, there are lots of things we haven't discussed in this short article. We haven't discussed how to install your driver (you write something called an INF file), specific techniques for driver development with any of the models, or strategies for debugging your code. But we have to leave **something** to write about in future issues.

We hope the above has been useful, and provided a place to start. Happy driver writing!

## WDF File Object Callbacks... (Cont.)

(CONTINUED FROM PAGE 7)

the quick retrieval of the WDF File Object from a native File Object pointer:

- **WdfFileObjectWdfCanUseFsContext**
- **WdfFileObjectWdfCanUseFsContext2**
- **WdfFileObjectWdfCannotUseFsContexts**

The native File Object has two context areas that are free for use by the driver that completes the create request: *FsContext* and *FsContext2*. While this would seem a natural place for the Framework to store the WDF File Object, there are cases in which these fields are already in use by another driver in the stack or have special meaning. This then requires the Framework to maintain its own lookup table to convert the native File Object into a WDF File Object. Due to the fact that the Framework cannot know the meaning of *FsContext* and *FsContext2* for an arbitrary device stack, **WdfFileObjectWdf CannotUseFsContexts** is chosen by default.

Lastly, KMDF v1.9 adds a new value, **WdfFileObjectCanBe Optional**. Unlike the previous values for this property, this is a flag value that may optionally be combined with **WdfFileObjectWdfCanUseFsContext**,**WdfFileObjectWdfCanUse FsContext2**, or **WdfFileObjectWdfCannotUseFsContexts**. This again has a very specific purpose resulting from potential driver –to-driver communication cases.

All I/O in Windows is sent by way of a native File Object that represents an open instance of a particular device object. This means that **all** I/O requests have an associated File Object that references the target device for the I/O operation. However, we again have to deal with the case of driver–to-driver communication. For example, Driver A may choose to send a request to Driver B without a File Object. Or with a File Object that points to Driver C. Without this optional flag set, if Driver B calls the **WdfRequestGetFileObject** in either of these cases there will be a KMDF Verifier exception thrown. If it is unclear whether or not a driver falls into either of these categories, the driver should *not* specify this option as it may mask a serious issue.

### Common Object Attributes

As with all KMDF objects, the WDF File Object also supports the Common Object Attributes as defined by the **WDF_OBJECT_ATTRIBUTES** structure. These attributes are optionally supplied when calling **WdfDeviceInitSetFileObject Config** to apply the **WDF_FILEOBJECT_CONFIG** structure to a **PWDFDEVICE_INIT** structure. There are potentially a few surprises lurking in the options available here, so we'll go through each of the Common Object Attributes and describe each one as they apply for this particular object.

#### *EvtCleanupCallback*

The *EvtCleanupCallback* callback is common to all Framework objects and indicates that either the Framework or the driver has triggered the teardown of a Framework object. In the case of WDF File Object, the naming is potentially confusing due to the native File Object's use of the term, "cleanup."

The lifetime of a WDF File Object is tied to the underlying native File Object. Thus, teardown of the WDF File Object does not start until the native File Object has been destroyed. Therefore the WDF File Object's *EvtCleanupCallback* will not trigger until after *EvtFileClose*.

#### *EvtDestroyCallback*

The *EvtDestroyCallback* callback is closely related to the *EvtCleanupCallback*. While *EvtCleanupCallback* indicates that teardown of a WDF object has begun, the *EvtDestroyCallback* indicates that teardown is complete and the WDF reference count of the object has gone to zero. Again, do not confuse this with the native File Object reference count; in order for the WDF File Object reference count to drop to zero the native File Object reference count must already be zero. Assuming that the driver has not done anything exotic, such as calling **WdfObjectReference** on the WDF File Object, the *EvtDestroyCallback* executes immediately following the *EvtCleanupCallback*.

#### *ExecutionLevel*

The *ExecutionLevel* constraint is interesting with the WDF File Object. The Framework disallows any create requests arriving at IRQL >= DISPATCH_LEVEL, regardless of the *ExecutionLevel* provided. The underlying issue is that some portions of the native File Object are pageable, thus processing create requests at IRQL >= DISPATCH_LEVEL is not architecturally defined within Windows itself. If the driver is aware of the risks in processing create requests at elevated IRQL and has incorporated them into its design, there are other methods to handle this condition. We will introduce one of the methods at the conclusion of this article.

However, if the driver does *not* specify an *ExecutionLevel* constraint of **WdfExecutionLevelPassive**, it should expect its *EvtCleanupCallback* and *EvtDestroyCallback* to be callable at IRQLs greater than PASSIVE_LEVEL.

#### *SynchronizationScope*

The rules for *SynchronizationScope* on WDF File Object are not made entirely clear by the documentation, though are in fact quite simple. First, it is always correct to specify a synchronization scope of **WdfSynchronizationScopeNone**, which causes the Framework to provide no specific serialization of the WDFFILEOBJECT Event Processing Callbacks.

Next, it is **never** correct to specify a synchronization scope of **WdfSynchronizationScopeQueue** on a WDF File Object. Earlier, we mentioned that the WDF Request Object passed to the

(CONTINUED ON PAGE 21)

## WDF File Object Callbacks... (Cont.)

(CONTINUED FROM PAGE 20)
*EvtDeviceFileCreate* Event Processing Callback is unique in that it is not queue presented. By this we mean that the WDF Request Object has no associated WDF Queue Object, therefore it makes no sense to have a synchronization scope of queue. Attempts to create a WDF Device Object with WDF File Object Event Processing Callbacks set to **WdfSynchronizationScopeQueue** fail with STATUS_INVALID_DEVICE_REQUEST.

Lastly, **WdfSynchronizationScopeDevice** is a valid choice **only** if the parent WDF Device Object has a PASSIVE_LEVEL execution constraint, otherwise the Framework fails the attempt to create the device. While seemingly onerous, as mentioned previously, processing create requests at elevated IRQL does not necessarily make sense. If the Framework allowed a driver to specify a synchronization scope of device with no execution level constraint, the WDF File Object Event Processing Callbacks would always execute at raised IRQL.

### *ParentObject*
It is invalid to override this property for WDF File Object objects; The parent must always be the WDF Device Object being created.

### *ContextSizeOverride/ContextTypeInfo*
Standard options for providing per-object context are available for WDF File Objects.

### What About Queue Presented Creates?
As an alternative to providing an *EvtDeviceFileCreate* Event Processing Callback, it is possible to route create requests to a driver created queue by calling **WdfDeviceConfigureRequestDispatching**. In this case, the create request arrives at the queue's *EvtIoDefault* Event Processing Callback and is subject to the synchronization scope and execution level constraints applied to the queue. An *EvtDeviceFileCreate* Event Processing Callback may also be registered without error, though it will in fact never execute.

What is interesting about this option is that it does **not** impose the execution level constraint requirements of the *EvtDeviceFileCreate* Event Processing Callback. Therefore it is possible to route create requests to a queue with a synchronization scope of device and no execution level constraint, resulting in the driver's create request processing occurring at IRQL DISPATCH_LEVEL. In this case, the driver must be careful to not access any members of the native File Object that are pageable, such as the file name.

Another interesting point to note is that as a result of this method, the create request becomes queue presented. Thus, unlike handling create operations in your *EvtDeviceFileCreate* Event Processing Callback, create requests that are handled in *EvtIoDefault* as a result of request dispatching **will** have an associated WDFQUEUE.

Note that even if create requests are routed to a queue, cleanup and close operations are still only accessible via the *EvtFileCleanup* and *EvtFileClose* Event Processing Callbacks.

### Closing Remarks
We've found WDFFILEOBJECTs to be quite handy over the years in the drivers we've written. Hopefully you too now have a handle (get it?) on the mechanics of working with WDFFILEOBJECTs and can start using them effectively in your drivers.

Follow us!

## OSR'S CORPORATE, ON-SITE TRAINING
Save Money, Travel Hassles; Gain Customized Expert Instruction

We can:

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

To take advantage of our expertise in Windows internals, and in instructional design, contact an OSR seminar consultant at +1.603.595.6500 or by email at seminars@osr.com

## Windows Pool Manager (Cont.)

(CONTINUED FROM PAGE 9)

Indeed, as Windows has evolved, the pool allocator has become increasingly sophisticated in its management of pool. Thus, pool is now organized by NUMA Node as well as by individual CPU. The idea is that in order to minimize contention, each CPU will use its own pool region first and when it needs more memory it can get it from the NUMA node specific allocation. Some versions of Windows will level out kernel memory allocations in order to ensure uniform performance across the entire array of CPUs, while application specific memory is typically allocated from the local CPU's pool cache (it is not restricted to that CPU, but rather preferred by a given CPU in order to optimize performance).

### Pool Tags

When memory is allocated, a four-byte tag value can be specified. By convention this is usually just an array of four characters; in some circumstances the pool allocator will interpret the high bit of the tag to have special meaning.

For those performing crash analysis, this pool tag can help in figuring out what a given pool block represents based upon that value. For those writing drivers, picking unique pool tags is invaluable in tracking down and finding memory leaks as well as aiding in forensic analysis of crashes reported by test teams and maybe even customers.

The debugger uses a file called **pooltag.txt** in order to display diagnostic information about these tag values. On my development system this file is found in either C:\Program Files (x86)\Windows Kits\8.0\Debuggers\x64\triage or C:\Program Files (x86)\Windows Kits\8.0\Debuggers\x86\triage. On my system these two files are identical.

This file consists of a series of lines that include the tag value and then the corresponding "hint" that should be displayed by the debugger:

```
Irp  - <unknown>    - Io, IRP packets
```

*Note: if you were to look at the original code you would see a string like ' prI' in the call to* ***ExAllocatePoolWithTag****. That's because of the little endian nature of Windows platforms, so that the bytes appear in memory in the "correct" order.*

For someone using the debugger, this allows the use of this information for inferring the type of a given pool region. For example, if something is a device object, typically it will consist of an object header (**nt!_OBJECT_HEADER**) followed by the actual device object. Using this knowledge allows us to compute the address of the object and then supply that result to the **!devobj** command.

### Conclusion

A basic understanding of the Windows memory allocator can be helpful in understanding both the function of this critical OS feature as well as in simplifying debugging, particularly of OS components.

Follow us!

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.

Next presentation:

Dulles/Sterling, VA
24-28 March

For more information, visit www.osr.com/debug.html, or contact an OSR seminar coordinator at seminars@osr.com

## Peter Pontificates (Cont.)

(CONTINUED FROM PAGE 3)

to go old school and create sources and dirs Files and use the Windows 7 WDK. Later, if my client or I want to upgrade that driver, we're going to be stuck using the same ancient build environment or upgrading the driver project to a new, Visual Studio integrated, WDK … but then you lose XP support. You can't win, basically.

Note that the problem is similar (though not as bad) if you need to support Windows Server 2008 and later. You're stuck using the Windows 8 WDK. However, in this case – assuming I understand this right, I haven't successfully gotten it to work – you're **supposed** to be able to upgrade your driver project so that it works with both the Windows 8 WDK using VS 2012 and the Windows 8.1 WDK using VS 2013. That's not convenient, but it's not great either.

And don't even ASK me about trying to support a single binary that uses up-level features on OS versions where those features are available. This used to work. I know, I was part of the (then DDK, not WDK) team that helped spec and test it. But today? I sure haven't gotten it to work, I'll tell you that.

Last, I want to whine about my new friend, UMDF 2.0. It's a new Framework that's provided for the first time in the Windows 8.1 WDK. It's cool because it uses almost exactly the same syntax as KMDF 1.x – and that's just about the best idea since chicken and waffles. As a result, you can try running your driver in user-mode, and if you don't like its performance… you can move it to KMDF and run it in kernel-mode with a trivial amount of effort. Awesome! What's not to like, right? Well, there's this one thing: UMDF 2.0 is only supported on Windows 8.1 and later versions of Windows. So its use is going to be limited to very particular and special cases.

I know time marches forward, but as of today there are few companies who will target their drivers to only Windows 8.1 and later. I dunno… maybe for my birthday Microsoft will start supporting UMDF 2 on older operating systems. Now, that would be cool. Especially if they'll support it back to XP. Hey, if I'm going to dream, I might just as well dream big.

*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 |
|---|---|---|
| Kernel Debugging & Crash Analysis | 24-28 March | Dulles/Sterling, VA |
| Writing WDF Drivers | 28 April—2 May | Palo Alto, CA |
| Advanced WDF Drivers | 5-8 May | Palo Alto, CA |
| Developing File Systems | 13-16 May | Waltham, MA |
| Internals & Software Drivers | 23-27 June | Dulles/Sterling, VA |

## 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:

- 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.
- This was, in my opinion, the best and honestly, only class available to learn the subject material.
- The [Instructor] is a very knowledgeable windows driver engineer and has provided us with great training.  Everyone found the training to be extremely beneficial to our future project.
- 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.