The NT Insider

The only publication dedicated entirely to Windows® system software development

A publication by OSR Open Systems Resources, Inc. Not endorsed by or associated with Microsoft Corporation.

January—February 2011 Digital Edition Volume 18 Issue 1

The Wonderful World of Software Drivers

When you're new to the world of Windows driver development, nothing seems simple. Take, for example, the job of monitoring system activity or collecting information from kernel mode. You *might* expect this to be a relatively straight-forward project. And, it is… once you understand a few basics about the different types of software-only drivers that you can write.

Two Types

Software-only drivers (or just "software drivers" as they are most often called) are drivers that do not interact with device hardware. That is, they (a) do not claim hardware resources such as registers, ports, or interrupts, (b) do not manage the operation of a hardware device, and (c) do not attach to an already existing device stack that has a Function Driver. Software drivers are often referred to as "Kernel Services" because they provide non-hardware related functions, such as system monitoring or data collection. For example, if you need to monitor changes to the Registry or know which executable images are loaded into the system, you almost certainly would want to write a software driver.

There are two types of software-only drivers:

- Legacy-style software drivers
- PnP-aware software drivers

Legacy-style software drivers are based on the original, Windows NT, driver model. This is the model that Windows NT used before PnP and power management were introduced. Legacy-style software drivers are the most common, and the most simple, type of software driver. This type of software driver is appropriate for just about any task for which you would need a software-only driver and is therefore the type of software driver that we at OSR typically recommend people write.

(Continued on page 14)

2011 Seminar Schedule

Writing WDF Drivers 7-11 February, Boston/Waltham, MA
Kernel Debugging/Crash Analysis 14-18 February, Columbia, MD
Windows Internals & SW Drivers 7-11 March, Columbia, MD
Developing File Systems 14-17 March, Brussels, Belgium
Writing WDM Drivers 14-18 March, Santa Clara, CA
Developing File Systems 11-14 April, Boston/Waltham, MA

For more formation, visit www.osr.com/seminars.

Digital Edition FTW!

Thanks for your feedback on *The NT Insider*—Digital Edition. We appreciate the several hundred of you who took the time to write us.

The overwhelmingly positive feedback has led to a decision to continue publishing *The NT Insider* in PDF format. We're also *considering* providing an optional, paid, hardcopy subscription for those die-hards who want to continue to read *The NT Insider* on dead tree product.

Some answers to consistent requests:

Can you change the layout and have the articles "jump" less? Well, we *could*. But we really want to preserve the ability for folks to print the PDF themselves and read it in "traditional" format, at least for now.

My PDF reader doesn't handle your embedded links (requiring manual page continuation). Well, that does suck, but what you need is a better PDF reader. If you're reading this on an iPad, treat yourself to a copy of GoodReader from the AppStore.

Can we get back issues in PDF format? Maybe someday. It's a lot of work, and it's not like we make money from *The NT Insider*, you know? Know any big companies that'd like to sponsor *The NT Insider Collection, Digital Edition?*

Thanks again for *all* your feedback. We're happy to help the community, and glad you enjoy it!

The NT Insider™

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
Mark J. Cariddi
Scott J. Noone
OSR Associate Staff
Consultant At Large
Hector J. Rodriguez
Send Stuff To Us:
email: NtInsider@osr.com

Single Issue Price: $15.00

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

Inside This Issue:

Bash Those Bugs!

The WDK Community Bug Bash Rolls Along

Since its announcement five months ago, the bug bash has been proved exceptionally popular. Over 120 bugs have been filed, with about 20% of those *already* fixed and about half of the total bugs filed still pending.

As bugs are filed, they're validated by OSR. After validation, OSR's engineering team members submit the bugs directly to Microsoft. As bugs are resolved by Microsoft, OSR lets the original submitter of the bug know that the status has been updated. Bug status is kept up-to-date on OSR Online, where the community can review the submitted bugs – and Microsoft's progress in closing open issues – at any time.

Also, as bugs are validated awards are sent to the contributors. So the OSR staff has been kept busy mailing t-shirts, key chains, flash drives, and mouse pads.

The vast majority of bugs filed have been to correct errors or request clarification of the WDK documentation. Thankfully, the WDK doc team has been tremendously responsive, fixing problems (***thank you* WDK DOC TEAM!**). With any luck, those fixes will start showing-up in the latest online updates to the documentation soon. There have been plenty of bugs filed on the samples, header files, and build environments. And the WDK dev team has also been extremely responsive (***thanks* WDK DEV TEAM!!**). Unfortunately, we'll probably have to wait until Windows 8 to see the results of the WDK dev team's fixes.

Unless we decide to extend the Bug Bash, it is scheduled to end on 1 February 2011. At that time we'll be awarding the Big prizes: HP Mini Netbooks, Visual Studio with MSDN Ultimate subscriptions, free attendance at an OSR seminar, iPod Touch 32GB, and OSR Prize Packs. Pretty good swag, just for telling folks about bugs you encounter don't you think?

By any measure, the WDK Community Bug Bash has been a tremendous success. We're very grateful to our Cosponsor, ITT Defense & Information Systems. We're also *very* grateful for the assistance of the Microsoft WDK Team, specifically, the PM team, without whose help and support the Bug Bash would not have been possible.

File a bug, earn an award and a chance for prizes at:

www.osronline.com/page.cfm?name=bugbash

What OSR Students Say

Don't take our word for it. This is what students say about our seminars:

"I learned so much more in the week spent here than trying to learn on my own these past 4 years. I only wish I took the class back then. OSR continues to provide the best training experience we developers could wish for."

"It was an absolutely wonderful experience. [Instructor] knows the subject matter thoroughly, and can express the important points vividly and in easy to associate contexts. He is a very important player in the driver world, and made me feel very much welcome to this select community."

"Well organized. Well presented. Good food. No complaints ."

Peter Pontificates:

Pods of Fun

My skills in technology prognostication are long established and well known. About 15 years ago now, in a *Peter Pontificates* not unlike this one, I predicted that the Web was a passing fad and was sure to fall into disuse. After having so clearly established my ability to foresee the future, about 10 years ago I called for – and predicted – significant differentiation among x86 hardware platforms in the way that the memory and I/O buses were connected. I wrote that the way to get to a truly high performance computing experience was to, "bring us real map registers!"

It would be a mistake to assume that these demonstrations of technological prescience on my part are limited to times gone by. In fact, around the middle of 2010 I wrote, in part:

> *The iPad seems to me to be an iPod Touch with a case of Elephantiasis. I don't want one. I don't know why I **would** want one. In fact, I can **barely** conceive of why **anyone** would want one.*
>
> *I don't know about you, but I don't want to read the New York Times or a novel on an LCD display. And that means I don't want to read these things on an iPad.*
>
> *If the "pad" genre catches the imagination of the market, [w]e can be sure that scads of very similar devices will be brought to us by the clever folks in Taiwan, at prices that will be hard to beat. Not to mention, as I write this, Microsoft is reportedly readying "Courier" and Google is preparing an Android-based slate.*

My record for predicting technological trends thus remains intact.

The iPad clones from Taiwan are uniformly disappointing. "Courier" was cancelled. The first reasonable Android tablet (the Galaxy Tab) was released just before Christmas 2010 to less than stellar reviews. Eight months after its initial release, the iPad still lacks a serious competitor.

The original iPad is stunningly popular. According to the Wall Street Journal, 1 in every 9 people surveyed said they planned to give an iPad as a Christmas gift in 2010. To date, Apple has sold over 14 million iPads. FOURTEEN million. In just 8 months.

They haven't sold that many iPads because they suck and are useless. In fact, just a few months after writing the pontification quoted above *I bought an iPad*. I liked it so much that I subsequently gave iPads as gifts to several of the engineers here at OSR. Why?

Because the iPad is a darn good toy, that's why. It's an almost perfect device for consuming web content and media. It turns on instantly, connects to WiFi or 3G with ease, let's you surf the web, play games, read newspapers and ebooks, watch movies or TV shows, and listen to your music all on a single device. The battery lasts between 9 and 10 hours. If you travel, you get one item to carry along that is guaranteed to keep you amused on a long flight. If you sit at home on the couch, you have a friendly little device that lets you look things up while watching TV (grabbing your iPad and tapping something out doesn't seem nearly as anti-social as grabbing a laptop, opening it up, waiting for it to boot, logging in, and typing on the keyboard). If you want to read a newspaper with your morning coffee, you can do it with the iPad – even if that newspaper is published in another city or country.

(Continued on page 5)

Peter Pontificates...

(Continued from page 4)

In short, much to my surprise, the iPad is an excellent device that fills a new and unique niche. Sort of like the iPod before it.

So, aside from demonstrating how I can *occasionally* be wrong in matters of technology forecasting, why should I be pontificating on this topic here in *The NT Insider*? What does the iPad have to do with Windows driver development? Well, I'm writing this because the iPad presents a serious risk to Microsoft.

Oh, yes… I've heard the rumors about Microsoft slates and tablets and upcoming Windows support for the ARM processor. And I have no doubt that, one way or the other, Microsoft can knock-out some sort of a credible tablet device. But the thing that makes the iPad such a thoroughly satisfying device is the *entirety* of the experience: The ease of use of the interface; the ability to buy games and applications with just a touch of the slate; and, most importantly, the breadth of third-party application offerings.

So while I'd probably still like my iPad, I wouldn't be nearly as thrilled with it if I couldn't play Plants vs Zombies. I wouldn't be as excited if I couldn't read *The Wall Street Journal* each morning. And I wouldn't be nearly as pleased with my iPad if it didn't allow me to buy and read Kindle books.

I realize that the Windows Phone 7 app store ("Marketplace") could be a prototype for a similar feature for Windows-based tablets. However, it will take a pretty large installed base before significant numbers of application developers move their wares to a Windows-based platform.

Please understand the point I'm trying to make: It's not the dollars lost in iPad sales that represents the major challenge to Microsoft. It's the overall positive exposure to Apple that people get. You play with an iPad, you like it, and – consciously or unconsciously – you start to think "you know, maybe this Apple stuff isn't half bad." Before you know it, you've stopped thinking of Safari as some crippled piece of shite. Next, you start cursing web sites that provide Flash content, calling them "ignorant" and "behind the times" for not getting on the HTML 5 and H.264 bandwagon which (Apple assures us) are open standards and what we all really want.

Yup, you play with your iPad and before you know it you find yourself saying to your spouse: "You know honey, that Apple stuff is pretty good. Maybe you should look at getting a MacBook Pro as your next laptop instead of a Windows machine. I mean, 3K for a garden-variety laptop isn't so bad. It won't crash, it doesn't get viruses, and you don't have to deal with Windows update."

OK, perhaps I exaggerate a bit. But do you get my point? Apple doesn't just rack-up record technology profits from the iPad. It accrues monster positive mindshare and enthusiasm. That's what Microsoft will have trouble countering.

So, does that mean we should all abandon ship and start writing drivers for IOS? Hardly. But I do think these things present serious challenges to Microsoft for both the short-term and longer-term futures. Now, I'll tell you what Microsoft should do about it.

If I was president of the Windows Division at Microsoft (there would be a whole hell of a lot of things that I would change, but most specifically in terms of this topic) I would give every developer in the Windows division an iPad – as a gift, to keep, tax free. And I'd make them use it. I kid you not. I'd really do it. I think it'd be the best US$5M that Microsoft could spend at this point in time. Of course, you couldn't keep anything this large quiet. But what a message this would send to the industry, huh? It would say "We recognize that we've stumbled badly, but we're now deadly serious about regaining the lead in this technology area."

In any large organization, there's bound to be a ton of "not invented here" syndrome. There will also be plenty of people who, never having actually tried a competitor's product, nonetheless dismiss and mock it. Plus, you do actually have to live with a product and use it for a while to understand its strengths and weaknesses. And so it would be for the iPad. By giving every dev in Windows an iPad, it'd let them actually *feel* what the competition is doing. Heck, if they don't like their iPad, I'd let them sell it on eBay. Seriously. I bet darn few of them would end-up being sold.

The other thing I'd do is I would start designing and manufacturing Microsoft-branded hardware. Yes, yes, I know all about how important the OEMs (Dell, HP, and the like) are to Microsoft, and how Microsoft needs to be careful about treading on their turf. Those relationships can be managed.

What Microsoft sorely needs right now is a BIG win in the tablet space. They cannot settle for releasing a solution that's just OK. They need something that overtakes, and not merely imitates, the iPad. The only way they can control the end-to-end quality and experience, and get a device to market sufficiently quickly is by having total control of the solution. As an aside, I'd also build Microsoft designed and developed desktop and laptop systems. I firmly believe that there's once again a market for high-quality, *slightly* price premium, systems. Apple has proved this. Building their own hardware would also allow Microsoft to quickly and effectively address other emerging threats, such as that from Google's Chrome OS (hell… that could be another pontification in itself – the Chrome OS laptop is just a freakin thin client… OEMs have been making them for years using Windows Embedded Standard, which is a darn good product by the way. Why doesn't Microsoft smash this Chrome OS crap before it starts by driving a low-cost, high-quality, laptop running Windows

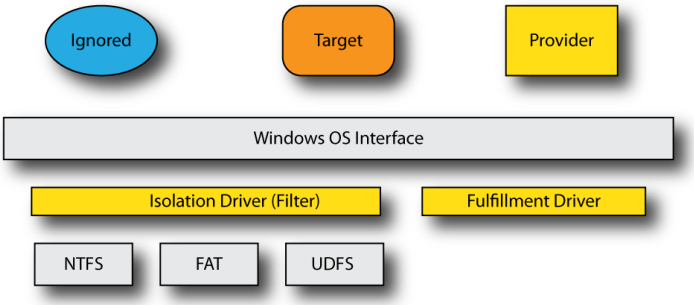
(Continued on page 23)

Getting Away From It All

The Isolation Driver (Part II)

In the first part of this series (*Getting Away From It All (Part I)*, http://www.osronline.com/article.cfm?article=560), we provided a high level introduction to our model of building an isolation driver. Since that time, we've received feedback from a number of people that have found this general model to be applicable to problems they are trying to solve.

In reviewing the original model we provided in Part I, we decided to logically separate out the functionality of the isolation driver into two distinct pieces: the *isolation* driver and the *fulfillment* driver.



By separating out the functionality into two logical pieces, we allow for the construction of a common isolation driver framework and its combination with a separate fulfillment driver. If your project doesn't require this sort of separation, you could combine their logical function together – we just decided to split this out because it allows us to build a common (core) isolation filter and allow someone else to construct a fulfillment driver, without requiring they understand the nuances of the isolation process.

Of course, our interest is in examining the issues involved in creating an isolation driver, not in the fulfillment driver (in other words, we don't really care where the data originates – that's the job of the fulfillment driver. We just need to handle the presentation of that data to the applications, plus handle the myriad of special cases and situations that might arise.

As it turns out, there are a substantial number of these issues, including:

- Asynchronous I/O
- Supersede, Delete & Truncate Streams
- Byte Range Locks
- Network File Systems
- Mixed 32/64 bit issues
- PNP/Dismount issues
- Filter-filter interactions
- Transactions

We'll cover these herein, with the balance of coverage for remaining issues to be addressed in Part III (multi-version support, compressed files, re-entrant create calls, reparse points, etc., etc.).

In this and subsequent articles we will cover these topics to further motivate our isolation driver example.

Asynchronous I/O

While most I/O operations are synchronous, for some applications as well as OS level callers, there are cases when the I/O operation can be completed asynchronously. This is an important issue for the isolation driver because it is a hybrid – not a "real" file system driver, but not a traditional filter driver either, as it controls its own cache.

Thus, we need to determine how asynchronous I/O should be implemented. But let's first start by making clear some of the basic rules of I/O that are true for all drivers in Windows:

- Any IRP can be implemented asynchronously by a driver. This is true even if the **IRP_SYNCHRONOUS_API** bit is set in the I/O stack location, or the **FO_SYNCHRONOUS_IO** bit is set in the file object. Corresponding to this is the rule for all other drivers in the system: if you call another driver, you must be prepared to handle asynchronous completion by the driver that you call, regardless of what options you set in the request. Some operations can be "wrapped" and wait (such as **IoForwardIrpSynchronously**). Be cautious, however, since some operations (directory change notifications and some FSCTL operations, for example) cannot be handled synchronously.
- Any I/O operation can be implemented synchronously by a file system driver. Even if the caller has not requested synchronous behavior, it can be implemented in such a fashion. A filter driver can generally do this as well.
- A driver may determine if the I/O operation is synchronous from the IRP by using **IoIsOperationSynchronous**. A filter driver may determine if the I/O operation is synchronous from the FLT_CALLBACK_DATA structure, by calling **FltIsOperationSynchronously.**

With respect to the isolation filter, we decided that the correct choice was to implement our I/O operations synchronously (at least as the general rule) and allow the *fulfillment* driver to implement asynchronous I/O. This works for us because the

(Continued on page 7)

Getting Away Part II...

(Continued from page 6)

fulfillment driver will need to handle asynchronous delivery in any case, such as if data delivery is being satisfied by a user mode service (using an inverted call, model, for example).

Supersede, Delete & Truncate

Subtleties that the isolation filter must handle are the various ways in which something can be deleted. These include the "destructive" create operations, specifically:

FILE_SUPERSEDE – this has the effect of deleting all the streams. Experientially, we have found that a supersede fails if any streams of the file are open, and this is something we need to keep in mind as we construct the isolation filter – depending upon the specific functionality that we require we *might* be able to defer this decision to the underlying file system. In particular, in pre-Vista systems we do not have *per file* contexts, and thus we either need to restrict ourselves to Vista (and more recent) systems or we will need to construct our own per file context tracking scheme. In addition, in all cases we should also keep in mind the mapped file cases (and particularly if our provider service and/or fulfillment driver are using streams for some or all of their functionality, which has the potential to complicate things). For our sample code, we will "keep this simple" but for your own isolation project you might need to address this.

FILE_OVERWRITE – this has the effect of truncating the data in the stream, as well as deleting all other streams when the main (default) data stream is overwritten. Unlike the supersede case, from what we have observed, the streams are deleted as they are closed. Again, for our sample, we will simply allow this to be handled by the underlying file system but you may wish to consider handling this in your own isolation driver project.

In addition, we have two ways in which a file can be deleted:
FILE_DELETE_ON_CLOSE – this create option has some interesting properties. First, it is tracked on a per-open instance (for a mini-filter, you can think of this as being "stream handle context" associated state). Thus, it is possible for a file to be opened multiple times, with one of them being **FILE_DELETE_ON_CLOSE**. When the specific handle is closed, this is converted into a request to "delete the file" and subsequent attempts to open the file will fail with **STATUS_DELETE_PENDING**.

IRP_MJ_SET_INFORMATION (Disposition) – this is the second way in which a file may be deleted. In this case, the deletion is an *intention* and can be "undone" until the file itself is closed. Further, there are some interesting issues in cleanup and close processing with respect to files that are delete pending, because they may be memory mapped – and in that case the file deletion cannot be processed until the **IRP_MJ_CLOSE** (and this might actually occur on a different **FILE_OBJECT** than the last handle close).

IRP_MJ_SET_INFORMATION (Rename) – this is a subtle case, but one special case of renaming a file is the "replace if exists" option within the rename. For isolation filters these events may need to be tracked and reported to the fulfillment component.

IRP_MJ_SET_INFORMATION (Hard Link) – this is the same case as for rename, just a different (and much rarer) operation. Not all file systems support hard links.

Note that here we've discussed "deleting the file." If anything besides the default data stream is opened in this fashion, only the stream itself will be deleted, not the entire file. Further, there are some subtleties involving streams that we are not fully exploring here.

One important point to understand here: it is not possible, within a filter driver, to know if the file has been deleted. Because deletion is an intention, a lower filter (e.g., an "undelete filter") can reverse the deletion. This behavior is invisible to anything logically "above" that filter (including our isolation filter).

Byte Range Locks

Byte range locks present an interesting issue for the isolation filter: because the data view to the application is different than the data view to the fulfillment driver (and the provider service) these must be handled by the isolation driver. In other words, do not assume that you can rely upon the underlying file system driver to "properly" implement these.

(Continued on page 16)

Let the Race Begin

Debugging Race Conditions

One of the situations in which post-mortem debugging is useful is for identifying race conditions – timing windows, in which the state of the machine changes between two points in the code execution, so that decisions made later in the code are incorrect because the state of the system has changed in some fundamental way.

In today's case, we have what looks like a potential candidate for this, although we can't really prove it – but then again, in my experience we never really do "prove" race conditions – we detect them, theorize them, change the code to eliminate them and hope they never come back.

Then again, I've seen a printf in code make the bug go away as well, so the frustrating part of these cases is that the solution is never really definitive.

In today's case, we have a fairly mundane Windows 7 box. I recently added 4GB of memory to this Windows 7 system, and shortly thereafter it crashed (bug check 0x3B.) Naturally, having this happen right after a memory upgrade did make me slightly suspicious so I took the opportunity to explore this particular crash in a bit more detail.

One nice change (since Windows Vista) has been that the machines are configured by default to create kernel summary dumps, so that there is more that we can do than simply a superficial analysis of the problem.

Naturally, I started with the usual **!analyze –v** and looked at its results (See *Figure 1*).

Oddly, the first thing that jumped out at me is that I had seen a similar crash recently (we use crash dump examples in our kernel debugging seminar). This reminded me that it is not uncommon to see "patterns" like this as you look at crash dumps (particularly random crash dumps like this one). It

(Continued on page 9)

```
2: kd> !analyze -v
*******************************************************************************
*                        Bugcheck Analysis                                    *
*******************************************************************************

SYSTEM_SERVICE_EXCEPTION (3b)
An exception happened while executing a system service routine.
Arguments:
Arg1: 00000000c0000005, Exception code that caused the bugcheck
Arg2: fffff8800106f141, Address of the exception record for the exception that caused the bugcheck
Arg3: fffff88005f45960, Address of the context record for the exception that caused the bugcheck
Arg4: 0000000000000000, zero.

Debugging Details:
------------------
PEB is paged out (Peb.Ldr = 00000000`7efdf018).  Type ".hh dbgerr001" for details
PEB is paged out (Peb.Ldr = 00000000`7efdf018).  Type ".hh dbgerr001" for details

EXCEPTION_CODE: (NTSTATUS) 0xc0000005 - The instruction at 0x%08lx referenced memory at 0x%08lx. The memory could not be %s.

FAULTING_IP:
fltmgr!TreeUnlinkMulti+51
fffff880`0106f141 488b4620        mov     rax,qword ptr [rsi+20h]

CONTEXT:  fffff88005f45960 -- (.cxr 0xfffff88005f45960)
rax=fffffaf7072f96b0 rbx=0000000000000000 rcx=fffffa8008e13318
rdx=fffffa8007e5b550 rsi=0000009d00000000 rdi=0000000000000000
rip=fffff8800106f141 rsp=fffff88005f46330 rbp=fffffa8008e13318
 r8=ffffffffffffffff  r9=ffffffffffffffff r10=ffffffffffffe4a
r11=0000000000000001 r12=fffffa8007e5b550 r13=fffffa8007e34684
r14=0000000000004000 r15=0000000000000000
iopl=0         nv up ei pl nz na po nc
cs=0010  ss=0018  ds=002b  es=002b  fs=0053  gs=002b             efl=00010206
fltmgr!TreeUnlinkMulti+0x51:
fffff880`0106f141 488b4620        mov     rax,qword ptr [rsi+20h] ds:002b:0000009d`00000020=????????????????
Resetting default scope

DEFAULT_BUCKET_ID:  VISTA_DRIVER_FAULT
BUGCHECK_STR:  0x3B
PROCESS_NAME:  CarboniteServi
CURRENT_IRQL:  0
LAST_CONTROL_TRANSFER:  from fffff8800106c460 to fffff8800106f141

STACK_TEXT:
fffff880`05f46330 fffff880`0106c460 : fffffa80`07a96920 fffffa80`07e5b550 fffffa80`07a96920 00000000`00000000 : fltmgr!
TreeUnlinkMulti+0x51
fffff880`05f46380 fffff880`0106cbe9 : fffff880`05f48000 00000000`00000002 00000000`00000000 00000000`00000000 : fltmgr!
FltpPerformPreCallbacks+0x730
fffff880`05f46480 fffff880`0106b6c7 : fffffa80`08b93c10 fffffa80`07ca8de0 fffffa80`07b402c0 00000000`00000000 : fltmgr!
FltpPassThrough+0x2d9
fffff880`05f46500 fffff800`02da278e : fffffa80`07e5b550 fffffa80`07dfa8e0 fffffa80`07e5b550 fffffa80`07ca8de0 : fltmgr!
```

Figure 1— Jumping Into !analyze-v

(continued next page)

Let the Race Begin...

(Continued from page 8)

isn't *exactly* the same, however, but it is intriguingly similar. See the stack trace from that crash in *Figure 2*).

While not identical, what stuck out in my head is that they are both (essentially) in the same piece of logic, suggesting to me that there might be a data corruption or race condition issue going on here (although our original theory for the crash that we use in the seminar is that it is actually a single bit error).

Patterns of this type can (and have) brought insight in the past. So let's see where this analysis takes me, and whether or not it matches anything from that previous analysis. One good point here is that I have some experience in walking the relevant data structures.

One other note about this particular machine: it's not used for development; it's used for video editing. As such, it isn't running anything other than "stock" software that one might find on a typical production computer system.

In this particular case we seem to have a garden variety NULL pointer dereference:

```
fffff880`0106f141 488b4620        mov     rax,qword
ptr [rsi+20h]
ds:002b:0000009d`00000020=????????????????
```

Thus, **RSI** is null in this case and the resulting user mode address is not valid. As typical in a case like this, we'll walk backwards in the code stream to try and figure out where this value originated, since it is often the case we can glean some understanding of what this code is trying to do. So *Figure 3* shows the code leading up to this point (via the **u** command).

(Continued on page 10)

```
FltpDispatch+0xb7
fffff880`05f46560 fffff800`02a918b4 : fffffa80`07e34010 fffff800`02d8f260 fffffa80`06d17c90 00000000`ff060001 : nt!
IopDeleteFile+0x11e
fffff880`05f465f0 fffff800`02d900e6 : fffff800`02d8f260 00000000`00000000 fffff880`05f469e0 fffffa80`08b93c10 : nt!
ObfDereferenceObject+0xd4
fffff880`05f46650 fffff800`02d85e84 : fffffa80`07c3fcd0 00000000`00000000 fffffa80`07a17b10 fffffa80`0a31e701 : nt!
IopParseDevice+0xe86
fffff880`05f467e0 fffff800`02d8ae4d : fffffa80`07a17b10 fffff880`05f46940 0067006e`00000040 fffffa80`06d17c90 : nt!
ObpLookupObjectName+0x585
fffff880`05f468e0 fffff800`02d1ee3c : fffffa80`08cf07e0 00000000`00000007 fffffa80`00001f01 00001f80`00f40200 : nt!
ObOpenObjectByName+0x1cd
fffff880`05f46990 fffff800`02a8b993 : fffffa80`0a31e7e0 00000000`00000000 fffffa80`0a31e7e0 00000000`7ef95000 : nt!
NtQueryFullAttributesFile+0x14f
fffff880`05f46c20 00000000`77320eba : 00000000`00000000 00000000`00000000 00000000`00000000 00000000`00000000 : nt!
KiSystemServiceCopyEnd+0x13
00000000`0121e778 00000000`00000000 : 00000000`00000000 00000000`00000000 00000000`00000000 00000000`00000000 : 0x77320eba

FOLLOWUP_IP:
fltmgr!TreeUnlinkMulti+51
fffff880`0106f141 488b4620        mov     rax,qword ptr [rsi+20h]

SYMBOL_STACK_INDEX:  0
SYMBOL_NAME:  fltmgr!TreeUnlinkMulti+51
FOLLOWUP_NAME:  MachineOwner
MODULE_NAME: fltmgr
IMAGE_NAME:  fltmgr.sys
DEBUG_FLR_IMAGE_TIMESTAMP:  4a5bc11f
STACK_COMMAND:  .cxr 0xfffff88005f45960 ; kb
FAILURE_BUCKET_ID:  X64_0x3B_fltmgr!TreeUnlinkMulti+51
BUCKET_ID:  X64_0x3B_fltmgr!TreeUnlinkMulti+51

Followup: MachineOwner
---------
```

Figure 1— Jumping Into !analyze-v

(continued from previous page)

```
3: kd> k
Child-SP          RetAddr           Call Site
fffff880`06678208 fffff800`02afcb19 nt!KeBugCheckEx
fffff880`06678210 fffff800`02a7cfee nt! ?? ::FNODOBFM::`string'+0x40edb
fffff880`06678370 fffff800`02a624ae nt!KiPageFault+0x16e
fffff880`06678500 fffff880`010f6373 nt!RtlDeleteNoSplay+0x2a
fffff880`06678530 fffff880`010f2238 fltmgr!TreeUnlinkNoBalance+0x13
fffff880`06678560 fffff880`0111046f fltmgr!TreeUnlinkMulti+0x148
fffff880`066785b0 fffff880`01110dfe fltmgr!DeleteNameCacheNodes+0x9f
fffff880`066785f0 fffff880`011202af fltmgr!PurgeStreamNameCache+0x8e
fffff880`06678630 fffff880`01117a30 fltmgr!FltpPurgeVolumeNameCache+0x7f
fffff880`06678670 fffff880`01110d4b fltmgr! ?? ::NNGAKEGL::`string'+0x1a04
fffff880`066786b0 fffff880`010f306a fltmgr!FltpReinstateNameCachingAllFrames+0x4b
fffff880`066786e0 fffff800`02a81516 fltmgr!FltpPassThroughCompletion+0x8a
fffff880`06678720 fffff880`01250f30 nt!IopfCompleteRequest+0x3a6
fffff880`06678800 fffff880`012dadfa Ntfs!NtfsExtendedCompleteRequestInternal+0x110
fffff880`06678840 fffff880`01249e0c Ntfs!NtfsCommonSetInformation+0xef1
fffff880`06678920 fffff880`010f023f Ntfs!NtfsFsdSetInformation+0x11c
fffff880`066789a0 fffff880`010ee6df fltmgr!FltpLegacyProcessingAfterPreCallbacksCompleted+0x24f
fffff880`06678a30 fffff800`02d5f49d fltmgr!FltpDispatch+0xcf
fffff880`06678a90 fffff800`02a7e153 nt!NtSetInformationFile+0x909
fffff880`06678bb0 00000000`775f012a nt!KiSystemServiceCopyEnd+0x13
```

Figure 2— Stack from Similar Crash

Let the Race Begin…

(Continued from page 9)

Given that there is a **ret** instruction two lines earlier, we scan backwards for a jump or branch to that location. This type of code sequence is typical of an **if/else** style statement (where there is a **return** in one of the two code blocks).

```
fffff880`0106f124 7518            jne     fltmgr!
TreeUnlinkMulti+0x4e (fffff880`0106f13e)
```

At this point it certainly looks like there might be a logic bug here: the **test** instruction does a bitwise AND of the **RSI** register with itself:

```
fffff880`0106f121 4885f6          test    rsi,rsi
```

…and then conditionally jumps based upon whether or not that value is zero :

```
fffff880`0106f124 7518            jne     fltmgr!
TreeUnlinkMulti+0x4e (fffff880`0106f13e)
```

Thus, this tests the value in **RSI** to see if it is zero. If it is, we jump down and use it, a condition that certainly does not seem to make much sense here. When I saw this it made me question if I was interpreting the instruction sequence correctly. Of course, a sanity check would be, "Well, if it weren't zero, having it change to zero within the CPU would suggest a hardware problem of some sort."

The encouraging insight here is that this certainly doesn't look like it is related to the new memory added to the system.

Thus, the next logical step was to look and see, "So, where did the value in **RSI** originate?" That's also present in this instruction stream:

```
fffff880`0106f11e 488b31          mov     rsi,qword
ptr [rcx]
```

So if we look at the value in RCX:

```
2: kd> dq @rcx l1
fffffa80`08e13318  fffff8a0`0d106d58
```

Lt is peculiar – this is not a zero value. This could be due to the fact that not all registers are saved, although the fact we see **RCX** is non-zero suggests that it is saved (values that are not saved on the x64 platform show as zero in the trap frame and context record).

Thus, this suggests a few possibilities:

1. The value captured here is not correct;
2. The value loaded in the **RSI** register is incorrect (ergo, a CPU problem of some sort);
3. The value to which **RCX** points has changed since we captured it in **RSI**.

The hardware error theory seems rather unlikely, so let's focus on the other two for a bit.

If we look at the data type of **RCX** we can see some useful information (*Figure 4*, next page).

So this block of pool seems to include this structure. On x64 systems the pool header is 16 bytes long:

```
2: kd> dt _POOL_HEADER
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
```

(Continued on page 11)

```
fltmgr!TreeUnlinkMulti:
fffff880`0106f0f0 fff3            push    rbx
fffff880`0106f0f2 55              push    rbp
fffff880`0106f0f3 57              push    rdi
fffff880`0106f0f4 4883ec30        sub     rsp,30h
fffff880`0106f0f8 33ff            xor     edi,edi
fffff880`0106f0fa 488be9          mov     rbp,rcx
fffff880`0106f0fd 4883faff        cmp     rdx,0FFFFFFFFFFFFFFFFh
fffff880`0106f101 0f840c010000    je      fltmgr!TreeUnlinkMulti+0x123 (fffff880`0106f213)
fffff880`0106f107 4c89642458      mov     qword ptr [rsp+58h],r12
fffff880`0106f10c 4c8be2          mov     r12,rdx
fffff880`0106f10f 4983f8ff        cmp     r8,0FFFFFFFFFFFFFFFFh
fffff880`0106f113 0f85eb450000    jne     fltmgr! ?? ::FNODOBFM::`string'+0x504 (fffff880`01073704)
fffff880`0106f119 4889742450      mov     qword ptr [rsp+50h],rsi
fffff880`0106f11e 488b31          mov     rsi,qword ptr [rcx]
fffff880`0106f121 4885f6          test    rsi,rsi
fffff880`0106f124 7518            jne     fltmgr!TreeUnlinkMulti+0x4e (fffff880`0106f13e)
fffff880`0106f126 488bdf          mov     rbx,rdi
fffff880`0106f129 488b742450      mov     rsi,qword ptr [rsp+50h]
fffff880`0106f12e 488bc3          mov     rax,rbx
fffff880`0106f131 4c8b642458      mov     r12,qword ptr [rsp+58h]
fffff880`0106f136 4883c430        add     rsp,30h
fffff880`0106f13a 5f              pop     rdi
fffff880`0106f13b 5d              pop     rbp
fffff880`0106f13c 5b              pop     rbx
fffff880`0106f13d c3              ret
fffff880`0106f13e 488bdf          mov     rbx,rdi
```

Figure 3— Walking Backwards (u command)

Let the Race Begin…

(Continued from page 10)

To properly compute the start of the structure we need to take the start address of the pool block and add 0x10 to it (to account for the pool header):

```
2: kd> dt fltmgr!_STREAM_LIST_CTRL
fffffa8008e13270+10
   +0x000 Type           : _FLT_TYPE
   +0x008 ContextCtrl    : _FSRTL_PER_STREAM_CONTEXT
   +0x030 VolumeLink     : _LIST_ENTRY
[ 0xfffffa80`07043c60 - 0xfffffa80`08e03390 ]
   +0x040 Flags          : 0x211 (No matching name)
   +0x044 UseCount       : 4
   +0x048 ContextLock    : _EX_PUSH_LOCK
   +0x050 StreamContexts : _CONTEXT_LIST_CTRL
   +0x058 StreamHandleContexts : _CONTEXT_LIST_CTRL
   +0x060 NameCacheLock  : _EX_PUSH_LOCK
   +0x068 LastRenameCompleted : _LARGE_INTEGER 0x0
   +0x070 NormalizedNameCache: _NAME_CACHE_LIST_CTRL
   +0x080 ShortNameCache : _NAME_CACHE_LIST_CTRL
   +0x090 OpenedNameCache: _NAME_CACHE_LIST_CTRL
   +0x0a0 AllNameContextsTemporary : 0
```

And the value in **RCX** is an offset into this structure:

```
2: kd> ? @rcx-fffffa8008e13280
Evaluate expression: 152 = 00000000`00000098
```

Thus, this corresponds to something in the **OpenedNameCache** field. By dumping the **_NAME_CACHE_LIST_CTRL**, we can see what exactly that is:

```
2: kd> dt _NAME_CACHE_LIST_CTRL
fltmgr!_NAME_CACHE_LIST_CTRL
   +0x000 NameFormat       : Uint4B
   +0x008 List             : _TREE_ROOT
```

This in turn makes a bit of sense given the name of the function in which the fault occurs (ergo, we are manipulating a tree structure of some sort).

In expanding the _TREE_ROOT structure we see:

```
2: kd> dt fltmgr!_TREE_ROOT
   +0x000 Tree             : Ptr64 _RTL_SPLAY_LINKS
```

Thus, I suspect that this is embedded in some larger data structure (and that structure is in turn linked into this tree). So let's see what displaying the contents of the tree root tells us:

```
2: kd> dt _NAME_CACHE_LIST_CTRL @rcx-8 /b
fltmgr!_NAME_CACHE_LIST_CTRL
   +0x000 NameFormat       : 2
   +0x008 List             : _TREE_ROOT
      +0x000 Tree             : 0xfffff8a0`0d106d58
```

We can then use this address (which, oddly enough is *not* zero, making this dump a bit unusual – after all, we just loaded this value into the **RSI** register and it *was* zero) to inspect this structure (See *Figure 5* below).

Now let's take this containing structure and dump it (*Figure 6*, next page).

(Continued on page 12)

```
2: kd> !pool @rcx
Pool page fffffa8008e13318 region is Nonpaged pool
 fffffa8008e13000 size:  150 previous size:    0  (Allocated)  File (Protected)
 fffffa8008e13150 size:   30 previous size:  150  (Allocated)  Io
 fffffa8008e13180 size:   50 previous size:   30  (Allocated)  VadS
 fffffa8008e131d0 size:   80 previous size:   50  (Allocated)  MmSd
 fffffa8008e13250 size:   20 previous size:   80  (Free)       Io
*fffffa8008e13270 size:   c0 previous size:   20  (Allocated) *FMsl
                Pooltag FMsl : STREAM_LIST_CTRL structure, Binary : fltmgr.sys
 fffffa8008e13330 size:  150 previous size:   c0  (Allocated)  File (Protected)
 fffffa8008e13480 size:   e0 previous size:  150  (Allocated)  NV
 fffffa8008e13560 size:  160 previous size:   e0  (Allocated)  Ntfx
 fffffa8008e136c0 size:  160 previous size:  160  (Allocated)  Ntfx
 fffffa8008e13820 size:   20 previous size:  160  (Free)       FIPc
 fffffa8008e13840 size:   50 previous size:   20  (Allocated)  VadS
 fffffa8008e13890 size:  160 previous size:   50  (Allocated)  Ntfx
 fffffa8008e139f0 size:   c0 previous size:  160  (Allocated)  FMsl
 fffffa8008e13ab0 size:  160 previous size:   c0  (Allocated)  Ntfx
 fffffa8008e13c10 size:   c0 previous size:  160  (Allocated)  FMsl
 fffffa8008e13cd0 size:   10 previous size:   c0  (Free)       FIPc
 fffffa8008e13ce0 size:   60 previous size:   10  (Allocated)  Io
 fffffa8008e13d40 size:  160 previous size:   60  (Allocated)  Ntfx
 fffffa8008e13ea0 size:  160 previous size:  160  (Allocated)  Ntfx
```

Figure 4— Checking the Data type of RCX

```
2: kd> !pool 0xfffff8a0`0d106d58
Pool page fffff8a00d106d58 region is Paged pool
 fffff8a00d106000 size:  130 previous size:    0  (Allocated)  Ntfo
 fffff8a00d106130 size:  140 previous size:  130  (Allocated)  MPsc
 fffff8a00d106270 size:   10 previous size:  140  (Free)       .tFs
 fffff8a00d106280 size:   40 previous size:   10  (Allocated)  NtFs
 fffff8a00d1062c0 size:  110 previous size:   40  (Allocated)  ;oNm
 fffff8a00d1063d0 size:  190 previous size:  110  (Allocated)  .Mfn
 fffff8a00d106560 size:  4d0 previous size:  190  (Allocated)  .tff
 fffff8a00d106a30 size:   40 previous size:  4d0  (Allocated)  NtFs
 fffff8a00d106a70 size:  180 previous size:   40  (Allocated)  {Mfn
 fffff8a00d106bf0 size:   c0 previous size:  180  (Allocated)  oIcs
 fffff8a00d106cb0 size:   40 previous size:   c0  (Allocated)  MmSm
 fffff8a00d106cf0 size:   40 previous size:   40  (Allocated)  .tFs
*fffff8a00d106d30 size:  190 previous size:   40  (Allocated) *FMfn
                Pooltag FMfn : NAME_CACHE_NODE structure, Binary : fltmgr.sys
 fffff8a00d106ec0 size:  140 previous size:  190  (Allocated)  APsc
```

Figure 5— Digging Further

Let the Race Begin...

(Continued from page 11)

To be honest, this structure looks a bit suspicious to me – the "creation time" doesn't seem plausible, and that _FLT_INSTANCE address definitely does not look valid. So I decided to poke at the structure in a bit more detail, as shown in *Figure 7*.

```
2: kd> dt fltmgr!_NAME_CACHE_NODE fffff8a00d106d40
   +0x000 Type             : _FLT_TYPE
   +0x008 ProvidingInstance: 0x00000081`00000000 _FLT_INSTANCE
   +0x010 CreationTime     : _LARGE_INTEGER 0x3f`00067e77
   +0x018 TreeLink         : _TREE_NODE
   +0x050 NameInfo         : _FLT_FILE_NAME_INFORMATION
   +0x0c8 UseCount         : 1
```

Figure 6—Dumping the Structure

This looks somewhat valid to me, actually. The name for the volume looks to be properly set up, although several of the name components seem to be suspect – almost as if this structure is being initialized or torn down.

At this point I put together my working hypothesis: that there is a race condition present in this code somewhere, most likely in the initialization code. That still doesn't explain what seems to be a logic issue here. I'm still left with more questions than answers – but having *two* crashes in the same general area, on two different machines, with two radically different usage profiles does suggest there is something interesting going on here.

Why do I propose a race condition here? Because I see information in the crash that is inconsistent – the contents of a register are NULL, but the memory location from which it was loaded indicates it should be non-zero. I cannot tell

(Continued on page 13)

```
2: kd> dt fltmgr!_NAME_CACHE_NODE fffff8a00d106d40 /b
   +0x000 Type             : _FLT_TYPE
      +0x000 Signature        : 0xf204
      +0x002 Size             : 0x176
   +0x008 ProvidingInstance : 0x00000081`00000000
   +0x010 CreationTime     : _LARGE_INTEGER 0x3f`00067e77
      +0x000 LowPart          : 0x67e77
      +0x004 HighPart         : 63
      +0x000 u                : <unnamed-tag>
         +0x000 LowPart          : 0x67e77
         +0x004 HighPart         : 63
      +0x000 QuadPart         : 270583365239
   +0x018 TreeLink         : _TREE_NODE
      +0x000 Link             : _RTL_SPLAY_LINKS
         +0x000 Parent           : 0xfffff889`0d106d58
         +0x008 LeftChild        : 0x0000001d`00000000
         +0x010 RightChild       : 0xfffff835`0d17eb28
      +0x018 TreeRoot         : 0xfffffaff`08e13318
      +0x020 Key1             : 0xfffffaf7`072f96b0
      +0x028 Key2             : (null)
      +0x030 Flags            : 0x14000
   +0x050 NameInfo         : _FLT_FILE_NAME_INFORMATION
      +0x000 Size             : 0x78
      +0x002 NamesParsed      : 0
      +0x004 Format           : 2
      +0x008 Name             : _UNICODE_STRING "\Device\HarddiskVolume2\UsIrs\???"
         +0x000 Length           : 0xa4
         +0x002 MaximumLength    : 0xa6
         +0x008 Buffer           : 0xfffff8a0`0d106e10  "\Device\HarddiskVolume2\UsIrs\???"
      +0x018 Volume           : _UNICODE_STRING "\Device\HarddiskVolume2"
         +0x000 Length           : 0x2e
         +0x002 MaximumLength    : 0x2e
         +0x008 Buffer           : 0xfffff8a0`0d106e10  "\Device\HarddiskVolume2"
      +0x028 Share            : _UNICODE_STRING "--- memory read error at address 0x000000e8`00000000 ---"
         +0x000 Length           : 0
         +0x002 MaximumLength    : 0
         +0x008 Buffer           : 0x000000e8`00000000  "--- memory read error at address 0x000000e8`00000000 ---"
      +0x038 Extension        : _UNICODE_STRING "--- memory read error at address 0x000000f3`00000000 ---"
         +0x000 Length           : 0
         +0x002 MaximumLength    : 0
         +0x008 Buffer           : 0x000000f3`00000000  "--- memory read error at address 0x000000f3`00000000 ---"
      +0x048 Stream           : _UNICODE_STRING "--- memory read error at address 0x00000041`00000000 ---"
         +0x000 Length           : 0
         +0x002 MaximumLength    : 0
         +0x008 Buffer           : 0x00000041`00000000  "--- memory read error at address 0x00000041`00000000 ---"
      +0x058 FinalComponent   : _UNICODE_STRING "--- memory read error at address 0x000000d9`00000000 ---"
         +0x000 Length           : 0
         +0x002 MaximumLength    : 0
         +0x008 Buffer           : 0x000000d9`00000000  "--- memory read error at address 0x000000d9`00000000 ---"
      +0x068 ParentDir        : _UNICODE_STRING ""
         +0x000 Length           : 0
         +0x002 MaximumLength    : 0
         +0x008 Buffer           : (null)
   +0x0c8 UseCount         : 1
```

Figure 7— And Finally...Ahem.

Let the Race Begin...

(Continued from page 12)

exactly when that data structure was added to this structure, but the choice seems to be "this is a CPU bug" or "this is a logic bug in this driver." Of the two, in my experience the latter is far more likely than the former.

Hopefully, we'll see more of these crashes so that we can find a pattern as to what is happening and broaden our analysis. It is also distinctly possible if we're seeing this issue in "the real world" the filter manager team in Redmond has seen many more cases of this and have resolved the issue. We'll be watching for more of these – perhaps *you* have a crash like this one you'd like to share with us.

Software Drivers...

(Continued from page 1)

Some people mistakenly believe that all Windows drivers need to be PnP and Power Management aware. While this is generally true for drivers that support hardware, it is not true for software-only drivers. Legacy-style software drivers are in no way deprecated and are in fact still 100% supported by Windows. Think about it: The purpose of supporting PnP is to allow a driver to respond to the dynamic arrival and departure of a device. The purpose of Power Management is to allow a driver to participate in system power state transitions (such as the transition to sleep or hibernate) and to manage a device's power state. Therefore, unless you're writing a software-only driver that needs to be aware of power -state transitions (which would be a rare thing) a legacy-style software driver is exactly the type you want to write.

Of course, it *is* possible to write a software-only driver that is aware of PnP and Power Management events. This type of driver, referred to as a PnP-aware software driver, can be written using either the WDM or WDF models (though the WDF model is certainly most highly recommended). PnP-aware software drivers are "root enumerated", that is they are started by the PnP Manager, and exist in their own unique branch of the PnP device tree. Unlike their legacy-style cousins, PnP-aware software drivers work very much like typical hardware drivers: They receive the full complement of PnP and Power requests and are required to handle those requests just like a driver that supports hardware. The PnP and Power Managers don't make any special concessions for a driver just because it's root-enumerated and is not associated with any hardware.

Which Type to Choose

There are advantages and disadvantages to each type of software driver. The driver type you choose to implement will depend both on your needs and on the development models with which you are familiar.

Legacy-style software drivers are unquestionably the simplest type of software driver to write. However, writing this type of driver does require that you have some knowledge of standard Windows driver architecture. You'll need to code a DriverEntry entry point and a dispatch entry point for each IRP major function you want to support. You'll need to understand the different transfer types (Direct, Buffered, and Neither I/O), and be comfortable dealing directly with IRPs (to do such things as retrieve parameters from the IRP's I/O Stack Location) and I/O completion. None of these things is difficult, of course. But if your experience lies primarily in the realm of WDF, you might view having to learn these additional concepts as a bit of an annoyance.

As mentioned previously, the alternative to developing a legacy-style software driver is to write a software driver that is PnP-aware. If you're already familiar with KMDF, this might be the easiest option for you. The driver you write will be exactly like any other KMDF driver: You code a DriverEntry entry point that calls WdfDriverCreate, and an EvtDriverDeviceAdd entry point that creates your WDFDEVICE and one more WDFQUEUEs, which in turn contain a selection of I/O Event Processing Callbacks. The WDFQUEUEs present WDFREQUESTs to the I/O Event Processing Callbacks, and you process these requests as appropriate. You may choose, or not, to handle power management events using the usual WDF mechanisms. Getting your power management code "right" isn't likely to be a major effort, because KMDF will handle most of the details of this typically onerous task for you.

There *are* reasons to choose to write one type of software-only driver over the other, aside from just familiarity with the development model. These reasons have to do with the work that you need your software driver to perform.

(Continued on page 15)

NEW SEMINAR—Windows Internals for Forensic Analysts

Based on feedback from students and managers, OSR is in the process of organizing a new seminar covering topics of interest to those in the field of forensic analysis for information security and cyber warfare. This new Windows internals presentation includes hands-on lab time where attendees can "get their hands dirty" implementing solutions that explore functionality and solicit data from a Windows system.

A tentative outline is currently available at www.osr.com/forensics.html, and we expect to be available to present this seminar in a private, on-site format beginning in Q2 2011.

Software Drivers...

(Continued from page 14)

One of the most interesting characteristics of legacy-style software drivers is that they live in a "parallel universe" to that inhabited by drivers that are PnP/Power aware. Thus, a legacy-style software driver isn't merely excused from obeying the standard rules that apply to drivers that support PnP/Power, it actually lives in an environment where *no PnP or Power Management exists*. This means that even if a legacy-style software driver supplies handlers for PnP or Power IRPs, it will never receive those IRPs because the driver executes within an environment in which these IRPs are not supported. So if you need to write a software driver that's aware, for example, of changes to the system's power state a legacy-style software driver won't do the job – You'll need to write a PnP-aware software driver.

It's important to realize that even though legacy-style software drivers live in a "parallel universe", this does *not* limit their ability to interact with PnP- or Power-aware drivers. Legacy drivers can send requests to and receive requests from any other driver in the system (hardware or software-only, PnP-aware or legacy). Also, legacy-style drivers *can* be informed of the arrival and departure of classes of PnP devices by registering a callback using IoRegisterPlugPlayNotification.

Software Driver != Filter Driver

One common misconception about Windows software-only drivers is that they are the same as filter drivers. While filter drivers don't typically claim any hardware resources and almost never interact directly with hardware, they *do* attach to a device stack that contains an FDO. In that stack, they monitor, manage, or modify the operation of the underlying device. Filter drivers are almost always loaded via the PnP process and often need to be aware of the power state of the device stack in which they reside. A filter driver typically needs to understand the details of the hardware operations that take place within its stack. As a result, a filter driver is a lot more like a device driver than a software-only driver.

Because most software-only drivers are legacy-style drivers, an extra note about using legacy-style drivers as filter drivers is in order here: Legacy-style drivers are not well-suited to act as filter drivers for a PnP-aware device stack. Even if you can contrive to get a legacy-style software driver inserted at the top of (or, somehow, within) a stack of PnP drivers, this is unlikely to be a reliable or supportable configuration. Why? Well, that's a discussion that we'll reserve for when we discuss more about writing filters.

Go To It!

There are many things in Windows that are either difficult or impossible to control from User Mode. In addition, collecting certain kinds of data is often much easier in Kernel Mode. Let's say you want to control which programs get executed on a system, or you want to monitor all write operations to the Registry. In this case, a software-only driver is exactly what you need!

You might be surprised to learn that an example legacy-style software driver is provided with the Windows Driver Kit. Look under \src\general\registry for an example of a driver that filters Registry operations. It's not a particularly simple (or particularly well-written) example but it does demonstrate the Configuration Manager monitoring functions, including some advanced use of transactions.

So, that's the scoop about Windows software drivers. For most monitoring and reporting tasks, you'll almost always want to choose to write a simple legacy-style software driver. They're fully supported by Windows, and they're entirely immune from having to deal with the pain that is PnP and Power Management, because they live in an environment where PnP/Power does not exist. Of course, if you're already familiar with KMDF or you need to be aware of system power state transitions, you *can* write a software-only driver using that model. Either way, your job should be relatively simple now that you understand your options.

Happy software-driver writing!

Getting Away Part II...

(Continued from page 7)

If your isolation filter need only isolate *local* file systems, this is not a difficult task to achieve – you can simply use the FsRtl or Flt functions suitable to the task (see **FltInitializeFileLock** or **FsRtlInitializeFileLock** for more information.) Using these functions requires:

- Handling lock and unlock calls (both IRP and Fast I/O based)
- Enforcing byte range locks for non-paging I/O read and write operations. It is an error to enforce byte range locks for paging I/O (what this means is that for memory mapped files, byte range locks are advisory, but there is no mechanism for distinguishing user modifications to a mapped file versus write-backs from the cache)

We will discuss the issues for isolation of network file systems separately.

In considering byte range locks, keep in mind that we have two distinct "views" of the file and there is no reason to consider that byte range locking of one should interact with byte range locking of the other.

What does this mean for the isolation driver? If an application locks a region of the isolated view, it should do so against other applications accessing that isolated view. The fulfillment driver and provider service do not need to be aware of these byte range locks at all and, indeed, it is much simpler if we rely upon the byte range lock management of the underlying file system in such cases.

Thus, byte range locks are managed by the isolation driver with respect to the application(s) that are accessing the isolated view of the file (versus the native view of the file). This ensures the provider service will not "run into" byte range locks against the isolated view.

Network File Systems

An isolation filter can, in theory, work with any file system, local or network. For network file systems, however, there are a number of issues that will need to be considered for our isolation filter. For example, there is a fundamental question of "how do we interlock between applications running on different client systems?" There are essentially three ways we can suggest solving this problem:

- Refuse to allow multiple client accesses to the isolation view of the file. This is certainly the simplest implementation but will lead to different application behavior than the native access to the file. For example, Microsoft Word "detects" when a file is already in use by using shared access to that file (and making a copy of the file in case where a conflict is detected.) This is lost if you refuse to allow multiple client accesses.
- Permit multiple client accesses to the isolation view of the file, relying upon the underlying file system to police this behavior. This presents a separate challenge, since we only have one file and we are now attempting to police sharing behavior for two different views of the file (the native and isolated views of the file). Unless two files (or two streams of the same file) are used to arbitrate this access, the combination of these accesses across the two files is unlikely to provide a satisfactory solution.
- Permit multiple client accesses to the isolation view of the file, relying upon the provider service (or perhaps the fulfillment driver) to properly arbitrate this behavior. In essence this becomes an implementation of a (simplified) distributed lock management scheme of some sort.

Beyond file sharing we can then decide how to handle byte range locks: using split ranges (so with a single file on the remote, you would use 0-4EB for isolation view locks, and 4-8EB for native view locks, for example,) or extending the distributed lock management to actually implement range locking for the isolation view (and then simply relying upon the native file system to handle the byte range locks on the native view of the file).

Oplocks are another complication here, because oplocks and byte range locks are frequently incompatible with one another – byte-range locks must be policed on the server, oplocks allow clients to cache data, which obviates the need for them to submit I/O operations back to the server. While it is the SMB redirector that uses oplocks for its caching policy, it does not make oplock state changes visible to filters above the redirector. This will require us to either disable oplocks (by taking out byte range locks, typically,) build an SMB network protocol filter (something we've theorized but never done), or end up breaking cache coherency between multiple clients.

For our prototype isolation filter, we will stick with a no-shared-access model, which obviates our concerns here. Those concerns are "real" and may be an issue for your own isolation driver project, in which case you will need to address these concerns beyond what we have done in our sample.

Mixed 32/64-bit Issues

Because we must coexist with 32-bit user applications in a 64-bit world, it is important to keep in mind that our isolation filter must properly handle these cases. The most significant issue here is the problem of structures containing HANDLE values. For a 32-bit application, these will be 32-bit values, while for a 64-bit application these will be a 64-bit value.

*NOTE: While we're discussing this issue, we should note that there is a known API bug: both **IoGetRequestorProcessId** and **FltGetRequestor ProcessId** return ULONG values, but process IDs are*

(Continued on page 17)

Getting Away Part II...

(Continued from page 16)

handles. It's a minor nit (since so far no system has had enough processes to get a value that overflowed 32 bits) but it does demonstrate how easy it is to mishandle 32-bit/64-bit support in Windows.

The most troublesome of these are the inclusion of handles in the I/O parameter block (**IO_PARAMETER_BLOCK** or **FLT_IO_PARAMETER_BLOCK**):

```
    //
    //  IRP_MJ_SET_INFORMATION
    //
    struct {
        ULONG Length;
        FILE_INFORMATION_CLASS POINTER_ALIGNMENT
FileInformationClass;
        PFILE_OBJECT ParentOfTarget;
        union {
            struct {
                BOOLEAN ReplaceIfExists;
                BOOLEAN AdvanceOnly;
            };
            ULONG ClusterCount;
            HANDLE DeleteHandle;
        };
        PVOID InfoBuffer;   //Not in
IO_STACK_LOCATION parameters list
    } SetFileInformation;
```

The presence of that HANDLE value within the structure does create some grief for us, since the actual size of this data value will depend upon whether or not this is a 32-bit process on a 64-bit OS. If it is, the HANDLE size of the process is *not* the same as the HANDLE size of the driver, and it is the driver's responsibility to accommodate this switch in sizes.

Indeed, there are a number of operations in which a handle is embedded, including:

- **IRP_MJ_FILE_SYSTEM_CONTROL** – this includes a number of FSCTL operations, including **FSCTL_MARK_HANDLE**, and **FSCTL_MOVE_FILE**.
- **IRP_MJ_SET_INFORMATION** – in addition to the parameters block that we mentioned previously, this also includes **FILE_RENAME_INFORMATION**, **FILE_MOVE_CLUSTER_INFORMATION**, and **FILE_LINK_INFORMATION**.

In such cases the isolation filter (and possibly the fulfillment driver) will need to implement proper logic to accommodate this difference in handle sizes. Our sample isolation filter will demonstrate this point when we get to it in a subsequent copy of this article.

PNP/Dismount Issues

If your isolation filter will deal with removable media or removable devices, you will need to handle mount and dismount issues as well as plug-and-play.

In both cases, the most complicated aspect is not just handling the events themselves, but serializing against those state changes along all *other* code paths. After all, the basic functionality during a device removal or media removal event is "relatively straight-forward." We need merely delete our data structures. However, if those same data structures are being used in any other code path, we cannot delete them.

In general, the simplest way to protect against this is to use something akin to an **IO_REMOVE_LOCK**. However, these are documented as only working if they are within a device extension. So, to protect any other structures we might have, we will need to essentially build our own, likely using either **ERESOURCE** locks or perhaps **FltInitializePushLock** (in general, we tend to avoid using push locks because they make debugging deadlocks that involve them vastly more difficult). Thus, any code path that uses one of your data structures that is destroyed down the dismount or device

(Continued on page 18)

Getting Away Part II...

(Continued from page 17)

removal paths will need to be protected. You can accomplish this by acquiring the lock (**IoAcquireRemoveLock** or **ExAcquireResourceSharedLite**) and releasing the lock (**IoReleaseRemoveLock** or **ExReleaseResourceLite**) when you exit the protected path. Then, when you need to tear down the data structure, you need to lock it in the appropriate fashion (see **IoReleaseRemoveLockAndWait** or use **ExAcquireResourceExclusiveLite**).

Filter to Filter Interactions

A substantial area of complication for any filter relates to filter -to-filter interactions. It is unrealistic to assume that we can enumerate all the sources of such interactions, or advise on how to avoid all of them. However, there are a number of things that we can do to help minimize them:

- Never assume you can ignore a seldom used feature. We've seen filters fail to handle all sorts of specialized conditions (open by file ID, reparse points, hard links, etc.) It is important to think through these cases because they will need to be addressed at some point.
- Always ensure your filter can co-exist with itself. For example, if you need to detect re-entrant calls, make sure you use a detection technique that would be compatible with itself (adding a prefix or suffix to the file name is an excellent example of such a technique that does not "stack" properly. ECPs on the other hand do stack, provided each filter uses its own ECP entry).
- Don't bypass filters below you. It's tempting at times but can trigger compatibility issues.
- Ensure you go to Plugfest. This is the best way to test against a number of other filters, meet other developers, not to mention find and fix problems in a "real world" environment.

Keep in mind, no matter how well you build your filter, interaction issues are a reality of life. Functional interference (e.g., data scanning logic filters versus compression/ encryption filters) cannot be eliminated and active filters change the behavior of the file system stack, complicating the environment.

Transactions

Few things can be more complicated to get right than transactions (particularly in a complex filter, such as a data isolation filter). The simplest thing to do in an isolation filter is refuse to allow transactional operations on isolated files – this is likely to be a "first stop" for a first generation implementation, but can also lead to specific application failures.

In our experience to date, transactions are only used by installer programs (including Windows Update) and our test programs. Transactions are not yet in mainstream application use; whether they will be or not remains to be seen, but it is logical to expect to see them in specific types of applications in the future.

Thus, it is important to at least consider them in a full-blown isolation filter (we are not going to address this in our sample isolation filter, but it is an issue that may require you address it in a commercial implementation).

Then the question is: how do you support transactions in an isolation filter? In fact, the model that we use within an isolation filter (split views) really is inspired by the model in which transactions are implemented in NTFS – by using separate views of the data ("data isolation") via the Section Object Pointers structure.

There is quite a bit more to transactions than simply supporting split views, however. If your underlying file system supports transactions, you can defer a large number of operations to the underlying file system (notably, those dealing with the "shape" of the name space, for example). Without such support it is unlikely that you will be able to easily build a transactional rename facility (for example) into an isolation filter without constructing your own persistent resource manager. Note: discussing the creation of a resource manager of any type is beyond the scope of this discussion.

Thus, if you can restrict yourself to data isolation, you can then simply treat the transaction as being a separate "view" of the data (albeit with some model for how you handle rollback and commit, perhaps by using the CLFS support within Windows Server 2003 and more recent).

Summary

I wish I could say we're done, but then we would be leaving out some really important issues to understand with respect to implementing an isolation driver. We'll wrap up discussion of these in Part III.

Analyst's Perspective

Analyzing User Mode State from a Kernel Connection

Analyzing user mode state from the kernel debugger appears to have become something of a black art. Some people swear it can't be done, others swear that it can't be done reliably, and a small few claim that they do it all the time without any problems. I'm here to say that, yes, it *can* be done and to grow that small few to the vast majority…

When it comes down to it, there are only three things that you need to understand in order to properly work with user mode state from a kernel debug connection. So, let's explore each of these.

The Virtual Address Space in Windows

Windows maintains two different virtual address spaces, the user virtual address space and the kernel virtual address space. In a standard x86 installation, this division results in the low 2GB of virtual memory being given to the *current user process* and the high 2GB being the kernel virtual address space.

The lower portion of the address space changes depending on the thread currently executing on the processor. The higher portion of the address space however is the same across *all* process contexts. Thus, the lower portion of the address space is *process context specific* whereas the higher portion of the address space is *process context independent*.

WinDBG and Process Context

Understanding the virtual address space in Windows is a critical point to any analyst who wants to inspect user mode state. What one has to realize is that the debugger can only use *one* process context at a time to translate virtual addresses. This means that if you want to inspect user state you must make sure that you have instructed WinDBG to use the correct process context for that state. Failure to do so will lead to access errors or, even worse, incorrect or misleading information being returned.

Also worth noting at this point is that the *.thread* command does not change process context by default, thus simply switching to a different thread context is not sufficient to change your process context.

WinDBG and the User Mode Loaded Module List

The user mode loaded module list is our final piece to understanding working with WinDBG and user mode state. Unlike in kernel mode where we have a single loaded module list that WinDBG keeps track of, WinDBG does *not* keep track of the user module list for each process. Instead, WinDBG keeps a single list that represents the user module list at the time of the last *.reload*. What this means for you is that any time you begin working with a new user mode state, you want to make sure you refresh the user module list so that it matches the process you are analyzing.

Inspecting User State in Practice

Given that we now have the foundation, let's put the pieces together and see some practical examples. I'll start off by breaking in to an idle system from a live kernel debug session and inspecting the current process context:

```
0: kd> !process -1 0
PROCESS 8055c0c0  SessionId: none  Cid: 0000    Peb:
00000000  ParentCid: 0000
    DirBase: 00319000  ObjectTable: e1002e40
HandleCount: 253.
    Image: Idle
```

It's the Idle process, which isn't much of a shock. The Idle process is interesting in that it's one of the two *system processes*, which are processes with no user mode state. After a *.reload* we can inspect the user module list with *lmu*:

```
0: kd> lmu
start    end        module name
```

Note that there are no modules on the user mode loaded module list, which makes sense considering the fact that this is a system process. However, in the *!process 0 0* output I see an instance of Notepad and I really want to set a breakpoint in that process:

```
PROCESS 863c22f0  SessionId: 0  Cid: 00bc    Peb:
7ffdb000  ParentCid: 05fc
    DirBase: 06c602c0  ObjectTable: e16718e8
HandleCount:  29.
    Image: notepad.exe
```

(Continued on page 20)

Analyst's Perspective...

(Continued from page 19)

In order to do that, I need to use WinDBG's *.process* command to switch to the Notepad process context. In a live debug session we also want to specify the */i* to inspect the process state *invasively*. This will require that we resume the target machine, after which the target will break in to the debugger in the correct process context:

```
0: kd> .process /i 863c22f0
```

You need to continue execution (press 'g' <enter>) for the context to be switched. When the debugger breaks in again, you will be in the new process context.

```
0: kd> g
Break instruction exception - code 80000003 (first
chance)
nt!RtlpBreakWithStatusInstruction:
8052b5dc int     3
```

From here, we should be able to inspect the current process and see that we're in the Notepad process:

```
1: kd> !process -1 0
PROCESS 863c22f0  SessionId: 0  Cid: 00bc    Peb:
7ffdb000  ParentCid: 05fc
    DirBase: 06c602c0  ObjectTable: e16718e8
HandleCount:  29.
    Image: notepad.exe
```

However, we still do *not* have any user modules on our loaded module list!

```
1: kd> lmu
start    end        module name
```

Remember, WinDBG caches the user module list from the last *.reload*, thus we're still using the original loaded module list from the Idle process. In order to get WinDBG to refresh the user loaded module list, we need to perform a *.reload* again. Though we can save a bit of time here by just instructing WinDBG to reload the user module list with *.reload /user*:

```
1: kd> .reload /user
Loading User Symbols
.......................
```

Now we can actually see some results when inspecting the user module list:

```
1: kd> lmu
start    end        module name
01000000 01014000   notepad    (deferred)
5ad70000 5ada8000   UxTheme    (deferred)
5cb70000 5cb96000   ShimEng    (deferred)
…
7c900000 7c9af000   ntdll      (pdb symbols)
```

At this point in the analysis, we are free to inspect user mode state or set breakpoints in user mode routines. However, be aware that setting a breakpoint in a DLL mapped into multiple processes will result in the breakpoint being set in *all* of those processes. Writes from the kernel mode debugger are not subject to copy-on-write, thus setting a breakpoint with *bp* will put an int 3 instruction in the shared physical page. You can see the results of this here:

```
0: kd> !process -1 0
PROCESS 863c22f0  SessionId: 0  Cid: 00bc    Peb:
7ffdb000  ParentCid: 05fc
    DirBase: 06c602c0  ObjectTable: e16718e8
HandleCount:  29.
    Image: notepad.exe
0: kd> bp ntdll!ntcreatefile
0: kd> g
Breakpoint 0 hit
ntdll!ZwCreateFile:
001b:7c90d090 mov     eax,25h
0: kd> !process -1 0
PROCESS 8612abe0  SessionId: 0  Cid: 0430    Peb:
7ffdc000  ParentCid: 02a4
    DirBase: 06c60160  ObjectTable: e15d5858
HandleCount: 1115.
    Image: svchost.exe
```

A process specific breakpoint can be your savior here though:

```
0: kd> bp /p @$proc ntdll!ntcreatefile
```

Though the breakpoint will still be set in all processes sharing the page, the process specific breakpoint will cause WinDBG to only break if the breakpoint is hit by the specified process. Here we use the *$proc* pseudo register, which always maps to the current process.

Note what happens now if we become interested in a different process, say VMWareUser.exe:

```
PROCESS 86182878  SessionId: 0  Cid: 06e0    Peb:
7ffde000  ParentCid: 05fc
    DirBase: 06c60220  ObjectTable: e16091e0
HandleCount:  87.
    Image: VMwareUser.exe
```

We do all of the same processing as above and then check the user module list:

```
0: kd> .process /i 86182878
```

You need to continue execution (press 'g' <enter>) for the context to be switched. When the debugger breaks in again, you will be in the new process context.

```
0: kd> g
Break instruction exception - code 80000003 (first
chance)
nt!RtlpBreakWithStatusInstruction:
8052b5dc int     3
0: kd> !process -1 0
PROCESS 86182878  SessionId: 0  Cid: 06e0    Peb:
7ffde000  ParentCid: 05fc
```

(Continued on page 21)

Analyst's Perspective...

(Continued from page 20)

```
    DirBase: 06c60220  ObjectTable: e16091e0
HandleCount:  87.
    Image: VMwareUser.exe

0: kd> lmu
start    end        module name
01000000 01014000   notepad    (deferred)
5ad70000 5ada8000   UxTheme    (deferred)
5cb70000 5cb96000   ShimEng    (deferred)
...
```

Note how it looks like Notepad is mapped into the VMWareUser.exe process. Clearly this is bogus, it's just WinDBG using the cached user module list from the last *.reload* performed. Because our analysis has brought us to a new user process, we will again need to perform a *.reload /user* to have our module list updated:

```
0: kd> .reload /user
Loading User Symbols
..............................
0: kd> lmu
start    end        module name
00400000 00537000   VMwareUser   (deferred)
10000000 10010000   sigc_2_0   (deferred)
5ad70000 5ada8000   uxtheme    (deferred)
5b860000 5b8b5000   NETAPI32   (deferred)
...
```

What About Crash Dumps?

If you try to perform a *.process /i* command from within a crash dump, you'll be greeted with an error:

```
0: kd> .process /i 898c9020
```

This operation only works on live kernel debug sessions due to the fact that the invasive switch requires that code actually execute on the target machine. Luckily, there is a way to force WinDBG to internally switch to a different process context without changing the state of the target. For that, we'll use *.process* with the */r* and */p* switches. In addition to getting us into the correct process context, this will force a reload of the user symbol list:

```
0: kd> .process /r /p 86182878
Implicit process is now 86182878
.cache forcedecodeuser done
Loading User Symbols
..............................
```

Additionally, *.thread* also takes */r* and */p* switches to automatically switch the debugger to the correct process context for a particular thread. This is extremely helpful if you're moving around a full memory dump and would like to automatically have your process context set for each thread you inspect:

```
0: kd> .thread /r /p 863e5a60
Implicit thread is now 863e5a60
Implicit process is now 86182878
.cache forcedecodeuser done
Loading User Symbols
..............................
```

Seeing User State with !process and !thread

Last but not least, both *!process* and *!thread* take a flag value of 0x10, which causes the extension command to perform the equivalent of a *.process /r /p* for the appropriate process before displaying the call stacks of the threads. Thus, instead of this:

```
0: kd> !thread 86153418 f
…
nt!KiSwapContext+0x2f (FPO: [Uses EBP] [0,0,4])
nt!KiSwapThread+0x8a (FPO: [0,0,0]) (CONV: fastcall)
nt!KeWaitForSingleObject+0x1c2 (FPO: [Non-Fpo])
(CONV: stdcall)
win32k!xxxSleepThread+0x192 (FPO: [Non-Fpo])
win32k!xxxRealInternalGetMessage+0x418 (FPO: [Non-
Fpo])
win32k!NtUserGetMessage+0x27 (FPO: [Non-Fpo])
nt!KiFastCallEntry+0xfc (FPO: [0,0] TrapFrame @
ee59ed64)
WARNING: Frame IP not in any known module. Following
frames may be wrong.
0x7c90e4f4
```

Which aborts once entering user mode, you will see this:

```
0: kd> !thread 86153418 1f
…
nt!KiSwapContext+0x2f (FPO: [Uses EBP] [0,0,4])
nt!KiSwapThread+0x8a (FPO: [0,0,0]) (CONV: fastcall)
nt!KeWaitForSingleObject+0x1c2 (FPO: [Non-Fpo])
(CONV: stdcall)
win32k!xxxSleepThread+0x192 (FPO: [Non-Fpo])
win32k!xxxRealInternalGetMessage+0x418 (FPO: [Non-
Fpo])
win32k!NtUserGetMessage+0x27 (FPO: [Non-Fpo])
nt!KiFastCallEntry+0xfc (FPO: [0,0] TrapFrame @
ee59ed64)
ntdll!KiFastSystemCallRet (FPO: [0,0,0])
USER32!NtUserGetMessage+0xc
notepad!WinMain+0xe5 (FPO: [Non-Fpo])
notepad!WinMainCRTStartup+0x174 (FPO: [Non-Fpo])
kernel32!BaseProcessStart+0x23 (FPO: [Non-Fpo])
```

Black Art No More!

While there's always more to explore, hopefully this article serves to pique your interest and allow you to incorporate more user mode analysis into your kernel debugging sessions!

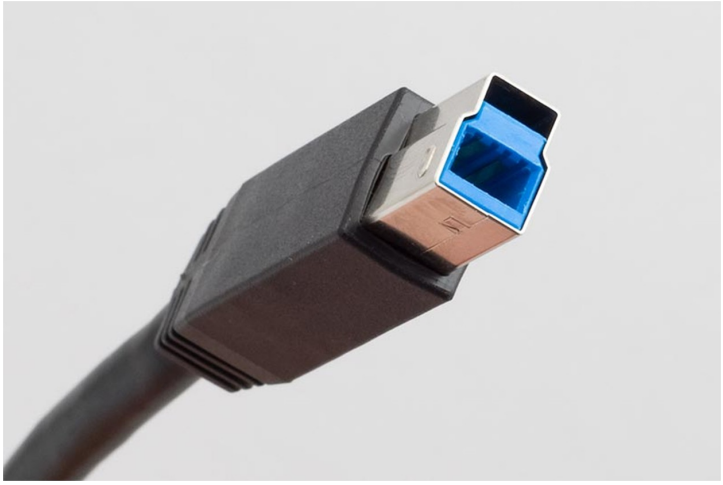
Analyst's Perspective is a column by OSR consulting associate, Scott Noone. When he's not root-causing complex kernel issues, he's leading the development and instruction of OSR's *Kernel Debugging* seminar. Comments or suggestions for this or future Analyst's Perspective columns can be addressed to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

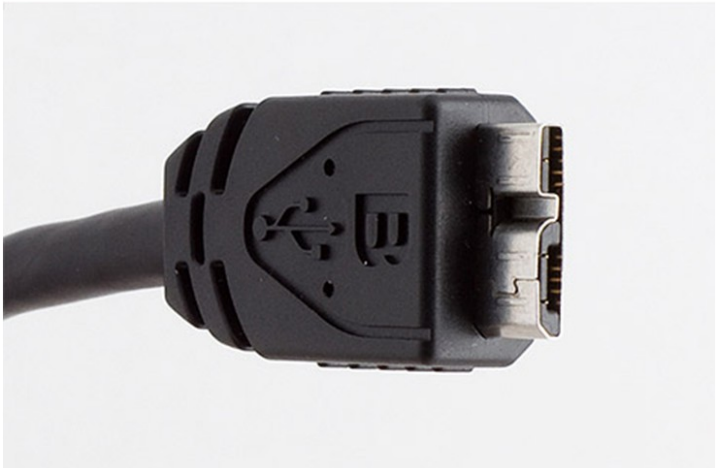
Go Blue!

In case you haven't seen them yet, USB 3.0 devices and host controllers are starting to appear in the world. USB 3.0 is capable of Super Speed (up to 5Gb/s). Aside from speed alone, USB 3.0 has several other advantages over previous versions of USB including reduced power consumption by devices when they are inactive.

The format for USB connectors A, B, and Micro-B connectors are shown on this page. USB 3.0 type A cables use connectors that are compatible with USB 2.0 devices. You can identify cables with type A connectors that are USB 3.0 capable by the color of the cable's "tongue" – cables with USB 3.0 compatible connectors are always blue.

While Windows does not yet support xHCI (the USB 3.0 Host Controller interface standard), there are third party Host Controller Drivers available. xHCI is compatible with older versions of USB, so your USB 1.1 and USB 2.0 devices should work just fine on a USB 3.0 controller. There are some "issues" about how Windows will be requiring USB 2.0 only ports to be wired on USB 3.0 controllers (regarding support for companion controllers) that we'll get into in more depth at another time. But, for now, suffice it to say that you probably do not want to install that shiny new xHCI controller into your Windows system unless you also plan to install and run a vendor-supplied (i.e. non-Microsoft) driver for that controller





Peter Pontificates...

(Continued from page 5)

Embedded into the market? What's *wrong* with them? And, when in the name of Gxd will Microsoft remove all the stupid restrictions on licensing Windows Embedded Standard and let people create systems for all sorts of devices? No viruses, no updates, no problems. Just boot it up, and it runs. But I digress).

The strategy of building Microsoft-branded equipment also fits very nicely with Microsoft's moves in opening their own retail stores. Have you ever been into an Apple store? If not, do yourself a favor: Find one, walk into it, and tell the first person you encounter that you're interested in finding out about Apple laptops. I'll be shocked if you don't find the experience exceptionally pleasant. Please, please, let the Microsoft retail stores be this good.

Maybe tablets are just a fad, and maybe a killer tablet from Microsoft is just around the corner. Maybe Windows 8 will be so new, and so exciting, that nobody will even remember that Apple is a technology company. Maybe Microsoft is, as I write this, readying a release of Windows 7 Embedded that'll run on a commodity laptop with WiFi and 3G. Maybe Microsoft will thereby consign Chrome OS to the same category as NeXTSTEP. Maybe. Maybe. Maybe.

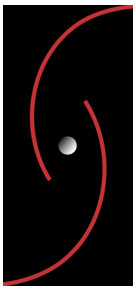
But, just in case… we already have people at OSR exploring how you write drivers and file systems for IOS and Android. And, no, I'm not kidding

Peter Pontificates is a regular opinion column by OSR consulting partner, Peter Viscarola. Peter doesn't care if you agree or disagree, but you do have the opportunity to see your comments or a rebuttal in a future issue. Send your own comments, rants, or distortions of fact to: PeterPont@osr.com.

Subscribe to The NT Insider Digital Edition

If you are new to The NT Insider (as in, the link to this issue was forwarded to you), you can subscribe at:

http://www.osronline.com/custom.cfm?name=login__joinok.cfm



OSR Open Systems Resources, Inc.
105 State Route 101A, Suite 19
Amherst, New Hampshire 03031 USA
(603)595-6500 ♦ Fax (603)595-6503

The NT Insider™ is a subscription-based publication

Subscribe to The NT Insider—Digital Edition

If you are new to The NT Insider (as in, the link to this issue was forwarded to you), you can subscribe at:
http://www.osronline.com/custom.cfm?name=login__joinok.cfm

New OSR Seminar Schedule!

Seminar	Dates	Location
Writing WDF Drivers (Lab)	7-11 February	Boston/Waltham, MA
Kernel Debugging & Crash Analysis (Lab)	14-18 February	Columbia, MD
Internals and Software Drivers (Lab)	7-11 March	Columbia, MD
Developing File Systems for Windows	14-17 March	Brussels, Belgium
Writing WDM Drivers (Lab)	14-18 March	Santa Clara, CA
Developing File Systems for Windows	11-14 April	Boston/Waltham, MA

Course outlines, pricing, and how to register, visit the www.osr.com/seminars!