ISSUE VOLUME
01/21
MAR-APR
2015

The NT Insider

A publication of OSR Open Systems Resources, Inc.

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

THE_NT_INSIDER_RELEASED
An attempt was made to access a ~~pageable~~ ~~pagable~~ pageable (or completely invalid)
article at an interrupt request level (IRQL) that is too high.  This is usually
caused by driver writers using improper time management or giving improper
project estimates.
If kernel debugger is available get stack backtrace. If kernel debugger is NOT
available, how are you seeing this message??
Arguments:
Arg1: 00000001, issue
Arg2: 00000021, volume
Arg3: 00000000, value 0 = issue contains Pontification, 1 = no Pontification
Arg4: 00000004, page which contains Peter Pontificates

Issue Details:
--------------

*** ERROR: Issue load completed but symbols could not be loaded for NTInsider.sys
ARTICLE_COUNT:  4

FAULTING_IP:
Page6!XperfIsHappiness

PROCESS_NAME:  KmdfAndUmdfHints.exe

PROCESS_ARGUMENT: PageNumber=8

ARTICLE_BUCKET_ID:  FILE_SIZES_IN_WINDOWS

ARTICLE_PAGE_NUMBER:  10


STACK_TEXT:
WARNING: Unwind information not available. Following page number may be wrong.
PageNumber ArticleTitle
00000012    FixYourOfflineSymbols

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

GET THE WINHEC EXTRA EDITION **HERE**!

Other Special Features

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

Looking for Free Technical Help from Peers?
OSR's NTDEV, NTFSD and WINDBG Lists

OSR hosts the most active, informative and well-moderated peer technical mailing lists on Windows systems software. Read/contribute to the lists via email, web, or even your favorite news reader.

NTDEV -- Windows System Software Developers List
This list is dedicated to the development of drivers for the Windows family of operating systems. Looking for Win32 or user-mode peer support? Not here. But if you've got a design, implementation or support issue with a Windows kernel-mode driver, this is the place to be. Inhabited by newbies, seasoned veterans, and even the Microsoft developers... Take the opportunity to learn from your peers to get over that particular development hump.

NTFSD -- Windows File System Developers List
Developing file systems or file system filter drivers? You need to be intimately familiar with the topics discussed on this list. Really...you can't afford not to partake of the information that is exchanged in this list.

WINDBG -- Windows Debugger Users List
Having trouble debugging a particularly thorny issue?? Do you find it impossible to make WinDbg do what you want?? This is the list for your issues.

Oh yeah, and when you want to vent off-topic…

NTTALK -- Extended Discussions About System Software
Hate/Love some feature in the Windows I/O subsystem??? Think Linux does it better/worse? Want to hold forth on why nobody ever will (or perhaps should) be able to write drivers in C#?? Why C++ is a plague on human kind?? This is the list for you. Free-flowing, rambling, sometimes even intelligent and thought provoking... that's what you'll find on NTTALK.

OSRHINTS: TECHNICAL TIPS FROM OSR VIA EMAIL

Not everyone has the time to keep up with various flavors of social media - and in some parts of the world, it's not even possible.

For such folks, OSR created the OSRHINTS mailing list, where we will duplicate our Twitter posts of technical hints, tips, tricks and other useful industry or OSR-related updates.

To options to join:

1. Send a blank email to:
   join-osrhints@lists.osr.com
2. Visit OSR Online and sign-up interactively:
   http://www.osronline.com/custom.cfm?name=listJoin.cfm

Get Social with OSR
Real -Time Updates

Follow us!

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

SAL Annotations: Don't Hate Me Because I'm Beautiful
It's time to give SAL annotations another try...
http://www.osr.com/blog/2015/02/23/sal-annotations-dont-hate-im-beautiful/

Would you Like Some Pi With Your Windows 10?
Bakers local to Redmond help announce tasty combination.
http://www.osr.com/blog/2015/02/02/like-pi-windows-10/

The Ferris Wheel is Heading Up: Microsoft Bringing Community Engagement
MSFT Loves Driver Devs?
http://www.osr.com/blog/2015/01/28/ferris-wheel-going-up/

Connected Standby: It's not Just for SOCs anymore
A primer in 939 words.
http://www.osr.com/blog/2015/01/14/connected-standby-just-socs-anymore/

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

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

WE KNOW WHAT WE KNOW

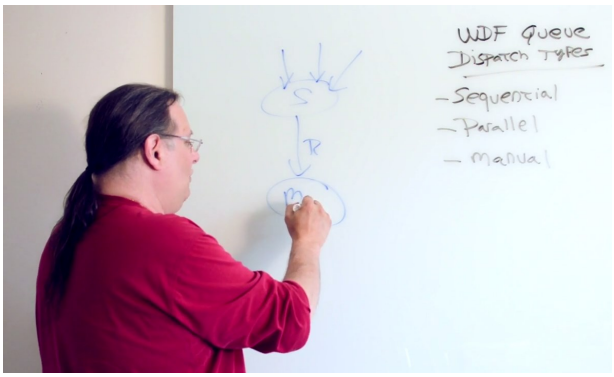
We are not experts in everything. We're not even experts in everything to do with Windows. But we think there are a few things that we do pretty darn well. We understand how the Windows OS works. We understand devices, drivers, and file systems on Windows. We're pretty proud of what we know about the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

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

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

Contact: sales@osr.com

Peter Pontificates:
Progress Means Moving Forward



I know that progress is a good thing. Progress is defined by Merriam-Webster as "gradual betterment," which is kind of a crappy phrase but it gets the meaning across nonetheless. So I **like** progress. Who doesn't, right? Everyone likes it when things get "better." Yay! As we say here in Boston (and we've said it a lot in the last few years) "Queue the duck boats, let's have a pahty."

But, just so we're all clear: **progress** is not the same thing as **change**.

By now, I should probably tell you WTF I'm talking about. OK, I'm talking about Windows 10. And lots of other application interfaces, like the ones on the iPad's WSJ app and the Weather Channel apps. But I really want to concentrate on talking about Windows 10. And, yes, I'm going to talk about the start menu. And in order to do that, I have to talk about Windows 8. And Windows 7. And Windows 95, which is where I'll start. Ha. Where I'll "start." Kinda like a joke.

The Windows 7 start menu traces its roots back to at least Windows 95. Maybe further. But when I got to 20 years back, I got bored and quit Googling. When this start menu structure was designed, people were using teeny tiny little screens. You clicked the start button, the start menu comes up, and you have just about everything you want at your fingertips. Organized in a space-efficient little strip, once you clicked "Programs" (which was updated to "All Programs" by Windows 7) you got an alphabetical list of folders that represented application categories. You opened the folders to find out what was in each group. Most of us figured this was pretty good. It worked reasonably well.

Many years later, the Windows 7 start menu gave us an updated and slightly more attractive version of this same Windows 95 start menu. And why not? It provided a perfectly reasonable, rational, method of doing stuff using smaller video screens. Except in 2009, when Windows 7 was released, more than 50% of users had screens that were larger than 1024x768. But, whatever.

Then along came Windows 8/8.1 and as we all know, the start menu disappeared. Instead, we got a start screen, which just about everyone will tell you is evil. "I can't use it", "I can't find anything", "It sucks", "I want Windows 7 back", "I hate Windows 8 because of the start screen"… you've heard all this, I'm sure. Heck, some of you probably say it. It's Internet wisdom, and thus, of course, it is true and not to be challenged.

The biggest complaint I hear from the "haters of the start screen" is that their stuff can't be located by category anymore like you could on the Windows 7 start menu.

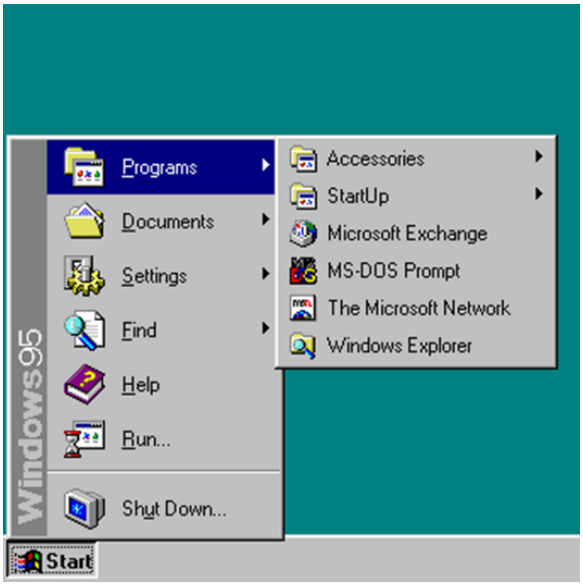


Figure 1—Love the Start Menu?

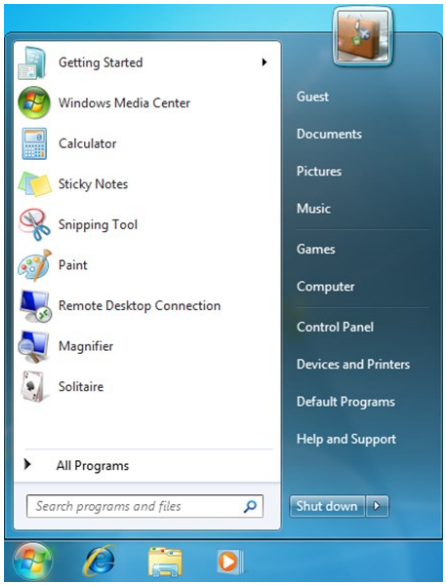


Figure 2—Windows 7 Start Menu… The Same, but Prettier

Well, I certainly agree that the ability to find your apps by category is vital. When I want to find "the command prompt thing for VS that has the ARM Phone tools" I should be able to look under "Visual Studio 2012" or "Windows Phone" something to find it.

Can't find it on Widows 8.1, can you! Nope! Because *the start menu is gone*. The **bastards**!

Well… on Windows 8.1, you go to the start screen (click the start screen button), then **click the down arrow on the start screen**. Voila! List of desktop apps, sorted by

(CONTINUED ON PAGE 5)

Peter Pontificates... (Cont.)

(CONTINUED FROM PAGE 4)

category. And since you have the whole screen, you can see all the apps under a given category. No more Windows 95 and Windows 7 style hiding stuff in "folders" that you have to open. Yay!

So ignore the fact that if you took two minutes to figure out how it works, you would plainly see that you can do everything (and more) from the Windows 8.1 start screen that you could do from the Windows 7 start menu… and with no increase in the number of mouse clicks. **And** tailored to take optimum advantage of the fact that we pretty much all have big screens these days and that those screens can quickly update, repaint, and show us different views.

Nope, ignore that. The Internet says the Windows 8.1 start menu sucks, so that's all most people need to know about it.

So, as you can see, I am definitely not one of the start screen haters. Well, let's be clear. I hate the Modern start screen, of course… but once you click the little down arrow and get the display you see in *Figure 3*, I'm loving it. Given that I sit at my system 60 or so hours a week, the fact that I can get work done with a minimum of annoyance is a good thing.

Therefore, regardless of what the Internet says, I would consider the Windows 8.1 start screen "progress" – or at least, not a step back in time.

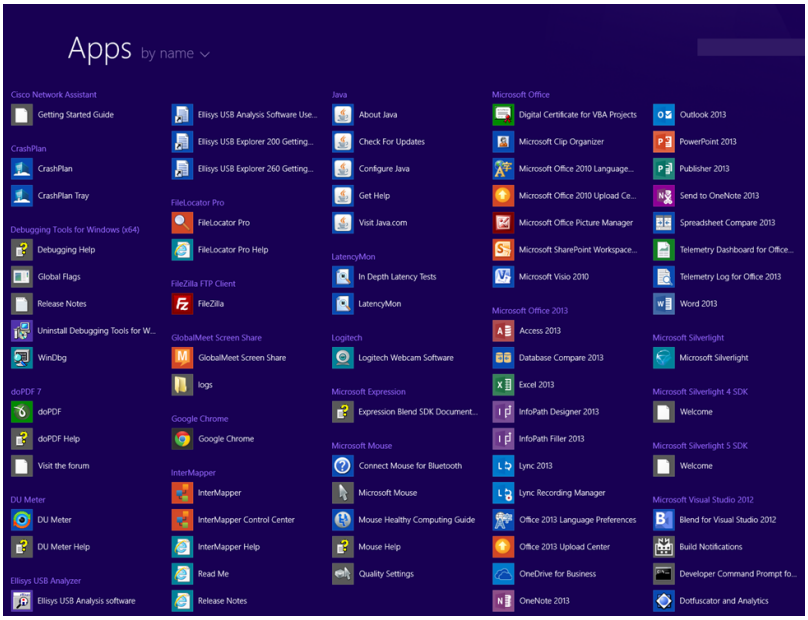


Figure 3 — Desktop Apps are Listed by Category, Just Like in Win7 Start, All Programs

But given that the Internet has been clamoring for the return of the start menu, and Microsoft's Team Nadella wants to appear receptive (perhaps even downright responsive!) to customer requests, the start **menu** is returning in Windows 10. Bye bye start **screen**. The result from (publically released) Build 9926 is shown in *Figure 3*.

Click on the start button, you get a little mini start screen. Then you click on "All apps", and what do you get? A kind of weird agglomeration of tiles, Modern app names, and folder names sorted in alphabetical order. So under "W" for instance, you see an icon for the "Weather" app, and a little folder icon that says "Windows Accessories" (click to open), oh… **there's** where Notepad is hiding. Hello Notepad!

(CONTINUED ON PAGE 31)

Happiness Is...
Xperf (and Custom Trace Events)

In 2012 a group of researchers published a paper titled, Awe Expands People's Perception of Time, Alters Decision Making, and Enhances Well-Being. The conclusion of the paper is that experiencing the feeling of awe might just make us happier, change our perception of time, and even make us more likely to volunteer (thus making the world a better place). This clearly indicates that we *need* more awe in our lives, we owe it to ourselves and to society as a whole.

Given this research, we figured it was time for another article about Xperf. No, not the French produced guinea pig food (which, according to the Amazon reviews, is awe inspiring in its own right, see *Figure 1)*, but the utility provided by Microsoft for performance analysis on Windows. We're always shocked to see how few people take advantage of what is one of the most awe inspiring tools you can get for free these days.

Figure 1

We discussed the basics of collecting traces back in 2010 with the article *Get Low - Collecting Detailed Performance Data with Xperf*. If you've never read that article please do that now as it should give you a good feel as to what Xperf is all about. Note however that article references an old GUI that has since been replaced by the **Windows Performance Analyzer** utility, which we'll be using in this article.

For this article, we're going to take a different approach than we did previously. We're going to create an incredibly trivial and overly simplistic example that demonstrates timer skew and see how we might use Xperf to analyze the cause of that skew. In addition, we'll show you how to inject your own, custom trace points into an Xperf analysis to help identify specific behavior in your driver.

The Example: A Timer That Fires Every 15ms…Or Not…

Did we mention that the example would be incredibly trivial and overly simplistic? Don't worry, the implementation will be even more trivial and simplistic. Our goal is to use Xperf to methodically extract information from a Windows machine exhibiting a particular behavior, not design a real solution to a real problem. Please keep eye rolls to a minimum and focus on the analysis!

Let's say we have a simple KMDF driver that requires a periodic timer that fires every 15ms, which we know to be **about** the granularity of the interval timer on Windows. We've heard that Windows is not a real time OS, but we choose to believe that things will be "close enough" and plow ahead with our implementation anyway. We're also good with the timer firing within 5ms either way of our deadline, so as long as it doesn't take more than 20ms between timer fires, we won't severely injure someone (much).

KMDF makes the creation of our timer callback simple, we just initialize the appropriate configuration structure and create the timer with **WdfTimerCreate:**

```
WDF_TIMER_CONFIG_INIT_PERIODIC(&timerConfig,
                               NothingEvtTimerFunc,
                               15);

    WDF_OBJECT_ATTRIBUTES_INIT(&objAttributes);

    objAttributes.ParentObject = device;

    status = WdfTimerCreate(&timerConfig,
                            &objAttributes,
                            &devContext->Timer);
```

Prior to starting the timer, we capture the current system time with **KeQuerySystemTime** so that we can check the elapsed time in the timer callback. We then start the timer with **WdfTimerStart**:

(CONTINUED ON PAGE 7)

Xperf… (Cont.)

(CONTINUED FROM PAGE 6)

```
KeQuerySystemTime(&devContext->LastTimestamp);

(VOID)WdfTimerStart(devContext->Timer,
                    WDF_REL_TIMEOUT_IN_MS(15));
```

In the timer callback, we calculate the delta between the current time and the previously recorded time and check to see if it exceeded 20ms (in 100ns units). If it does, we record a "miss". Otherwise, we record a "hit":

```
VOID
NothingEvtTimerFunc(
    _In_  WDFTIMER Timer
    ) {

    LARGE_INTEGER currentTime;
    LONGLONG delta;
    PNOTHING_DEVICE_CONTEXT devContext;


    devContext = NothingGetContextFromDevice(WdfTimerGetParentObject(Timer));

    KeQuerySystemTime(&currentTime);

    delta = (currentTime.QuadPart - devContext->LastTimestamp.QuadPart);

    if (delta > ((20 * 1000 * 1000) / 100)) {

        devContext->Misses++;

    } else {

        devContext->Hits++;

    }

    devContext->LastTimestamp.QuadPart = currentTime.QuadPart;

    return;


}
```

(CONTINUED ON PAGE 22)

KMDF and UMDF Hints
Our Top Five for WDF Driver Devs

If you're a developer who's been using WDF for a while, you're probably comfortable writing a KMDF or UMDF driver. And, if you're comfortable writing a driver, you're probably reasonably comfortable setting up and using the debugger to help you get that driver working.

But if you're like most of us, you could still benefit from a few tips on developing and debugging. If so, here are five hints that every WDF driver dev can use to make their life easier.

Hint 1: Set Your Symbol Search Path Properly – This is Always Step 1

If you've been working with WDF for a while, you probably already know this. But it's worth repeating: There's not much you can do, except get frustrated, if you attempt to debug your WDF driver without using the proper symbols. This includes not **only** symbols for your driver, but symbols for the Windows OS and HAL, and symbols for WDF as well. Neither WinDbg itself nor the WDF Kernel Debugger Extensions will work correctly until you have **all** the correct symbols loaded. Therefore, loading the right set of symbols is not an optional step. If you're going to use the debugger, you really must do it.

Fortunately, getting your symbols set up is easy. If the system on which you're running the debugger can access the internet, you can issue the command ".SYMFIX" followed by ".RELOAD" to the WinDbg command prompt. ".SYMFIX" sets your symbol search path to the Microsoft Symbol Server, and ".RELOAD" causes the symbols to be loaded. You can find a more detailed description (including an optional video, even) of how to properly setup your WinDbg symbols here in the OSR Developer Blog. If your debug environment is **not** able to access the internet (poor you!), check out the article in this issue of The NT Insider about how to setup your symbols .

Hint 2: Enable Windows Driver Verifier and WDF Verifier When Testing – Always

While you're working on your UMDF v2 or KMDF based WDF driver, always enable WDF Verifier on your test system. If you're writing a KMDF driver (or a WDM driver for that matter), also always enable Windows Driver Verifier. These tools will help you catch errors in your code much earlier than you would otherwise. They are not noisy, they do not generate spurious errors, and they are both tools that are entirely appropriate to use while your driver is actively under development and "not yet finished."

You can control Windows Driver Verifier using the tool **Verifier.exe** that is installed as a standard part of any Windows installation. As part of enabling Windows Driver Verifier for your driver, select **both** your driver **and** WDF01000.SYS (the Framework itself) to be verified. Enable **all** the options in Windows Driver Verifier, except IRP Logging and anything with "low resources simulation" in the name. If you don't interact directly with PoFx, you should probably leave "Power Framework Delay Fuzzing" off as well. After enabling Windows Driver Verifier on your test system, you'll have to reboot for your new Windows Driver Verifier settings to take effect.

When you enable Windows Driver Verifier for a KMDF driver, WDF Verifier is also automatically enabled with basic settings. You'll probably want to enable more checks in WDF Verifier than Windows Driver Verifier enables automatically.

You enable WDF Verifier and specify the specific options you want to use for your driver via settings under your driver's key in the Registry. For KMDF, specify your WDF Verifier settings under the following path:

(CONTINUED ON PAGE 9)

Our Top Five for WDF Driver Devs... (Cont.)

(CONTINUED FROM PAGE 8)

HLKM\System\CurrentControlSet\Services\<drivername>\Parameters\Wdf

For UMDF, the path is:

HLKM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\WUDF\Services\<drivername>\Parameters\Wdf

The values that we recommend for you to set are the following (all REG_DWORDs):

- **VerifierOn**: Set to a non-zero value, enabled WDF Verifier
- **Verbose:** Set to a non-zero value, causes additional logging to the WDF Log
- **VerifyDownLevel**: Set to a non-zero value to enable the most WDF Verifier checks, **not** just those that were current for the version of WDF against which your driver was built.
- **EnhancedVerifierOptions**: For KMDF only… Set to 1, ensures your driver's Event Processing Callbacks return at the same IRQL at which they are called.

There are other useful values that you can set as well, but the items listed above are a good start. Check the MSDN documentation for "Registry Values for Debugging WDF Drivers" for the full list.

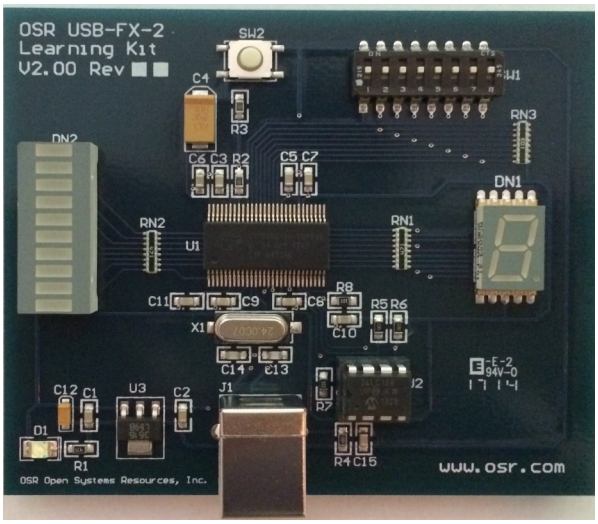
Note that WDF Verifier works with **both** UMDF V2 drivers as well as KMDF drivers. Remember to make these Registry entries on the test system, where your driver under test is running, not on the system you're debugging from! You'll need to restart the test system for the settings to take effect.

If you don't feel like playing with the Registry directly, the **WdfVerifier** utility (**WdfVerifier.exe**) ships in the \tools subdirectory of the 8.1 and later WDK. There's not much that's magic about this utility. It basically just makes the Registry changes for you, and therefore keeps you from having to remember the exact names of all the Registry values.

Regardless of how you do it, the important part of this hint is to **always** enable both Windows Driver Verifier (for KMDF drivers) and WDF Verifier (for KMDF or UMDF V2 drivers) on your test machine. Here at OSR, we often create our INF files to enable WDF Verifier in the Registry by default. If you do this, be sure to remember to delete those values before shipping your INF!

One more tip: To check what WDF Verifier settings are enabled for your driver, use the !WDFDRIVERINFO WinDbg command with a flag value of 1, as shown in *Figure 1 (page 17)*.

(CONTINUED ON PAGE 17)



On File Sizes
Logical and Physical File Sizes in Windows

One of the interesting challenges in building a robust isolation framework is the need to deal with file sizes. Our initial explorations into this area resulted from our work on our **File Encryption Solution Framework (FESF)**. However, we expect our work on isolation to form the basis of a generalized data virtualization framework that will eventually become one of our commercially available Solution Frameworks.

The fact that file sizes present a major challenge might surprise the casual reader. After all, one might wonder why file sizes would be any more complex than managing any other file attribute. However, because those file sizes are used by other parts of the OS, notably the VM system, understanding how these sizes are interpreted and the rules around that interpretation are important aspects of ensuring correct behavior.

From the perspective of a file system isolation filter, there are **six** different "file sizes" that we are managing. These six sizes fall into two groups:

- **Logical Sizes**: for the allocation, end of file ("file size" or EOF) and Valid Data Length (VDL)
- **Physical Sizes**: for the allocation, end of file and Valid Data Length

Logical sizes are what we present to the Virtual Memory system, and what applications perceive as the sizes of those files. What does each one represent?

Allocation Size represents the amount of disk space allocated. Typically, this is enough to ensure that we don't need to allocate more space during critical processing. For example, during paging write operations, there shouldn't be any need for allocating additional storage space. Of course, a file system (or isolation filter) **could** be written to handle this appropriately. For example, a file system might simply do *space reservation* and then actually pick the space when it's really required. The primary benefit to deferring allocation is that some access patterns can cause significant fragmentation, which is an issue for typical rotating media, but is actually not a concern for memory devices, such as NVME disks or SSDs.

End Of File is what we generally refer to as the "size" of the file. It indicates the number of bytes that this file contains. In isolation, a logical file might differ from the physical file because there is either additional data in the file (e.g., encryption headers, keys, additional state) or the data has different information density (e.g., compression or expanding encryption algorithms). Indeed, one of the primary justifications for constructing an isolation filter is to permit us to manage the file size independent of the physical storage. In that way, applications are blissfully unaware of the details of what we're doing within the isolation layer.

Valid Data Length represents the "high water mark" of data that has been **written** to the file. This differs from End Of File (which can be quite large) because it marks the region in which data has been written to the file.

Valid Data Length

Of these, Valid Data Length (VDL) is likely the most confusing. The reason Windows has this concept is to avoid an issue we refer to as *disk scavenging*. Traditional media file systems do not erase data contents on the drive when space is freed. Thus, to avoid exposing the contents of newly allocated space to the application, we track the region of the file that's been written by the application. Space beyond this point may be allocated, but the data contents are not returned to the caller – instead, they receive known safe data – usually zero filled memory, though nothing requires that specific pattern. In order to ensure that the region from VDL to EOF returns zeros, the file systems employ various techniques. In our experience we've seen four such mechanisms:

- Data between VDL and EOF is zeroed as part of allocation. This is the most resilient in terms of crash recovery semantics (there's never exposure of data inadvertently) and is also the most expensive in terms of performance.
- Data between VDL and EOF is zeroed as part of closing the file. This is the technique that FAT uses, and it does it during IRP_MJ_CLEANUP handling for the **last** open handle on the file. This may expose data in case of a crash, but it is generally much better at performance.
- The VDL of the file is stored as persistent information within the file information itself. This is the technique that NTFS uses. When a file is opened, NTFS can determine what the VDL of the file is and preserve correct behavior, even after a crash.

(CONTINUED ON PAGE 11)

Logical and Physical File Sizes (Cont.)

(CONTINUED FROM PAGE 10)

- Similar to FAT, VDL to EOF is zeroed on the last user handle close of the file, but with a journaling technique the file blocks are zeroed from VDL to EOF after a crash as well.

The VM system (Cache Manager and Memory Manager) components are concerned about VDL as well. This points to the fact that the valid data length is not only an "on disk" concept, but also an "in memory" concept. Thus, the VDL for data stored in cache **may differ** from the VDL as recorded on the disk itself.

This can lead to interesting behaviors in an isolation filter. For example, if an isolation filter skips over a disk region – saving space for some of its own meta data – the file system beneath the isolation filter will typically zero out that region. Normally it would do so via the VM system, which then triggers re-entrant I/O operations. If your isolation filter is not expecting to observe paging I/O operations in the midst of its own non-cached I/O operations to the underlying file system, it will likely not handle the situation properly.

Thus, in addition to the logical and physical sizes we have an "in memory" and "on disk" valid data length to consider! If you review the FASTFAT file system example in the WDK, you can see how FAT handles this case both during write (zeroing holes in the file) as well as during cleanup processing when the file is closed and it zeros the region from VDL to EOF.

I/O Rules

There are specific rules around the various file sizes and how they can be affected by I/O operations. To add even more complexity, these rules vary upon the context in which the I/O is being performed:

- User Cached I/O - in this case, writes can - and do - extend the size of the file. The file size may be reduced (*truncation*) via a call to either "set EOF" or "set Allocation". For NTFS and FAT, a user with appropriate privileges (SeManageVolumePrivilege) may set the VDL. *Note that there is no mechanism to **query** the value of the VDL, which turns out to be a problem for isolation filters that change file layout.* Thus, a write past EOF moves the VDL, EOF and possibly the Allocation size. The file system zeros data between the current VDL and the start of the new write. Filters will typically see such requests because the file systems use **CcZeroData** to perform this operation, which then issues a paging I/O (**IoSynchronousPageWrite**). But the zeroing is avoided when the user writes the entire region - even if it is written to the FSD **out of order**. This magic is possible because the FSD captures the identity of the lazy writer thread (though NTFS and FAT do this differently).
- User Non-Cached I/O - the operations must be sector aligned and sector sized, **except** at EOF, though the assumption is that we can use the remainder of the buffer, so it should be aligned to permit that as well. What we have noticed is that some applications (e.g., the xcopy utility) make worst case assumptions here: they sector size write and then truncate back to the correct EOF. Other applications behave differently. Note that extending writes are permitted in cases like this. Just like the cached case - but far more obvious - any holes in the file will be zero filled, unless the file permits sparse allocation.
- Paging I/O - all paging operations are done non-cached. There are special rules involved for paging operations

(CONTINUED ON PAGE 16)

Guest Article
Fix Your (Offline) Symbols!

Community Contributor
David Boyce

Scott Noone's blog post entitled *Setting the WinDbg Symbol Search Path* (you can read it here) contains some great advice. But his advice doesn't really apply if you're working in an isolated environment where you don't have direct access to Microsoft's Public Symbol Server. In these environments, there's typically some mechanism to logically or physically isolate your testing environment from the Internet. This isolation typically makes use of what's called an 'air gap' between your debugging environment and the Internet. If your organisation has this type of network isolation in place, presumably there are good reasons why it's there. If you still have to debug your driver… what do you do?

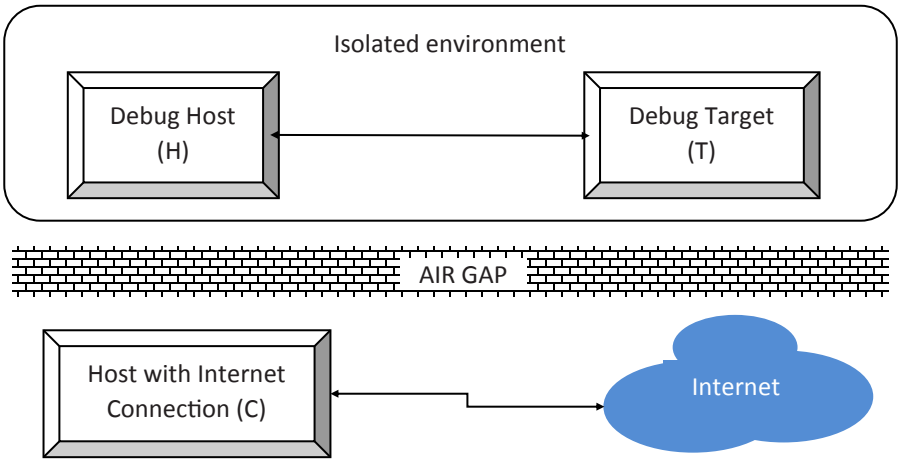


Figure 1 -- An "Air Gap"

The question becomes: Is it possible to make a debug target's (T's) kernel symbols available to your (isolated) debugging host (H)? Yes, if you can meet the following conditions:

1. You have access to the internet somewhere (machine 'C' above).
2. You have a means of transferring data in both directions across the air gap;
3. The means described in (2) is acceptable to your organisation.

Conditions (1) and (2) are basically technical issues, and the reader is assumed to be able to address them.

It's condition (3) that can get you into trouble if you don't pay careful attention. Some organisations will be concerned about bringing malware in from the outside, while others will be concerned about letting internal data out. These are both legitimate concerns, and any organisation which has set up an air gap will know what matters to them. Your job (or more) may depend on correctly following the rules laid down in this area.

SymChk with Manifest Files

The basic approach used here is described in the Microsoft Dev Center page **Using a Manifest File with SymChk.** But here I've also included the extra details needed to address the particular requirements of WinDbg kernel symbols.

In brief, the procedure is:

1. Create a symbol manifest file on the target (T, in *Figure 1*);
2. Transfer that manifest across the air gap to a machine (C, in *Figure 1)* with internet access;
3. Use the manifest to obtain the symbol files from the Microsoft Symbol Server;
4. Transfer the symbol files back across the air gap to the debug host (H, in *Figure 1*).

(CONTINUED ON PAGE 13)

Fix Your (Offline) Symbols!... (Cont.)

(CONTINUED FROM PAGE 12)

A few things specific to kernel-mode debugging are also important:

- Use the architecturally-appropriate version of SymChk on the target, i.e. 32-bit on x86 targets, and 64-bit on x64 targets;
- Use the name of the NT kernel for **/ie** argument;
- Avoid name resolution delays arising from attempts to access the public server (this is what often makes running SymChk offline painfully slow).

SymChk for the Target Architecture

You will need the following executables gathered together for the target platform:

- symchk.exe
- SymbolCheck.dll
- symsrv.dll
- symsrv.yes
- symstore.exe
- dbghelp.dll
- dbgeng.dll

You can find these in the Debuggers folder of your WDK installation on your debugging host machine (H, in *Figure 1*). You may need to install the WDK on a machine of the "other" architecture (perhaps a VM) to get that architecture's version.

You cannot run the x64 versions on a 32-bit OS for obvious reasons, and if you try to use the x86 versions on a 64-bit OS, you will see messages like

```
[SYMCHK] Can't load file "C:Windows\system32\hal.dll"
```

for each 64-bit executable.

I have two sets of these (one set for 32-bit systems, one for 64-bit) on a USB stick in separate folders so I can run the version appropriate to the target.

(CONTINUED ON PAGE 14)

Fix Your (Offline) Symbols!... (Cont.)

(CONTINUED FROM PAGE 13)

On the Debug Target

Open a command prompt and enter a command similar to:

```
symchk /om manifest.txt /ie ntoskrnl.exe
       /s c:\Users\Foo\DeskTop\Empty
```

Here, manifest.txt is name of the manifest file to be created ('/om') which will be transferred across the air gap outwards. The other arguments are explained below.

/ie Needs the Running Kernel's Name

It might seem obvious, but if you want the kernel symbols, you're going to need the name of the kernel executable, which is always in %SYSTEMROOT%\System32\ . In current versions of Windows, this name is (nearly) always **ntoskrnl.exe**, although I have also seen **ntkrnlpa.exe** on XP. In the following example, it's clearly ntoskrnl.exe, as the kernel is unlikely to be named ntprint.exe:

```
C:> dir /b %SYSTEMROOT%\System32\nt*.exe
ntoskrnl.exe
ntprint.exe
```

symchk's /ie option requires the name of a *currently* executing program. In the isolated environment scenario, there's a straightforward, if somewhat counter-intuitive, way to determine whether you have an incorrect executable name: if it reports

```
SYMCHK: FAILED files = 0
```

you've got the wrong name! A non-zero value indicates that it found the executable.

Of course, this isn't completely fool proof, since using the name of any currently executing program (not just the kernel) will produce similar reports. It does act as a useful sanity check, though.

When you're done, you'll know whether you've got the right symbols by the way WinDbg responds when you have completed the transfer.

Avoiding Name Resolution Delays on the Target

This is easy: create an empty folder and tell SymChk that's where to get its symbols from (/s):

```
mkdir c:\Users\Foo\DeskTop\Empty
symchk /om manifest.txt /ie ntoskrnl.exe
       /s c:\Users\Foo\DeskTop\Empty
```

By specifying an empty local folder, symchk doesn't attempt to contact any symbol server.

Data Going Out

At this point, you should have a manifest file which needs to be transferred across the air gap to the internet-connected machine (H, in *Figure 1*). The content of the file is a normal text file which can be opened by any editor. Each line represents an executable of some kind together with information indicating its version. Typically there are between 200-300 such lines in manifests wanting kernel symbols.

(CONTINUED ON PAGE 15)

Fix Your (Offline) Symbols!... (Cont.)

(CONTINUED FROM PAGE 14)

The vast majority of these lines represent executables which would be present on any Windows installation, and fetching their symbols does not represent any kind of security breach. However, it is possible that in some secure environments, there are executables listed for which the presence on the target system is confidential, and the attempt to fetch their symbols from the server would expose this. The simplest remedy (assuming that you don't actually need their symbols) is simply to remove lines referencing them from the manifest before transferring it.

Fetching the Symbols

On the host with internet access (C, in *Figure 1*), prepare an empty folder (e.g. C:\MySymbols) to receive the symbols and run the following command:

```
symchk /im manifest.txt
   /s SRV*C:\MySymbols*http://msdl.microsoft.com/download/symbols
```

Here, manifest.txt is the transferred manifest file which determines which symbols are to be obtained from the Microsoft public symbol server http://msdl.microsoft.com/download/symbols and placed below C:\MySymbols.

On success, C:\MySymbols contains the kernel symbols (and a number of others) that you need for debugging. The content of this folder needs to be transferred back across the air gap to the debug host (H, in *Figure 1*).

When using the /im option on a 64-bit machine, you can run either version of symchk.exe.

Fetch Errors

Don't be surprised to see some messages of the form

```
SYMCHK: <filename>      ERROR = Unable to download file. Error reported was 2
SYMCHK: FAILED files = <non-zero number>
```

even when <filename> is ntoskrnl.exe, provided the reported PASSED + IGNORED files is also non-zero.

For these purposes, such messages are (usually) benign; what's important is that C:\MySymbols is no longer empty and contains the kernel's PDB folder, e.g. ntkrnlmp.pdb\.

Data Coming Back

The symbols files are binaries - a mixture of .dll, .sys and .pdb files. You may wish to run a malware scanner over these before transferring them.

Finally...

When you've got your symbols back across the air gap, you have a local symbol cache at which you can point your WinDbg in the usual way. Good bug hunting!

David cut his programming teeth at Honeywell working on a middleware product written in assembler, then migrated to C and C++ product development on various platforms. He's come relatively recently to Windows Drivers, but hey - he always enjoys a technical challenge! He can be contacted at dboyce@becrypt.com.

Logical and Physical File Sizes (Cont.)

(CONTINUED FROM PAGE 11)

with respect to file sizes. Most important is that paging writes **do not** move the EOF. Similarly, they do not move the allocation of the file. As a result, we are limited in what we can do in this path. This should be OK - but it means that any EOF management must be done in the set information path. Note that Paging I/O operations may move the VDL.

In working with our isolation framework, we've found some surprising behaviors, which we'll attempt to capture in hopes they will help future developers in this area:

- The filter verifier has a bug where it will assert that your buffer is misaligned if you perform a sector aligned write, with a page aligned buffer, but a non-integral length **at end of file**. The file systems expect this case and all works as intended. For us, this meant that we had to avoid using FltWriteFile (or FltWriteFileEx if you're using the MDL passing interface) and instead explicitly built our own callback data structures for the writes – that path works correctly. The error that filter manager throws up is not only confusing (the buffer is aligned) but actually permitted by the file system.
- Paging writes to NTFS or FAT that are beyond the EOF **do not fail**. Instead, they return **STATUS_SUCCESS** and indicate zero bytes written. This was surprising to us and we spent quite a bit of time understanding the circumstances around this behavior. Much of this has to do with serialization and parallel file activity, as well as the manner in which the file systems handle files that have been deleted and/or truncated.
- Unlike allocation size and file size, **there is no mechanism for obtaining the VDL of a file** from the underlying file system. This is a complicating issue for an isolation filter that is independently tracking this, as opening an existing NTFS file might cause re-entrant file zeroing. It is possible to **set** the VDL of a file, though it requires the privilege as we mentioned earlier. That is because moving the VDL out without zeroing the file could expose information to the application. We decided against this approach and instead fell back to the FAT model: zeroing the file at final cleanup or anytime a "hole" is created within the file. That's unfortunate because it will create situations in which we perform I/O unnecessarily. Hopefully this is an oversight that Microsoft will rectify in a future version of Windows – that just doesn't help us now.

Our approach to isolation is also complicated when it comes to size management because we explicitly **do not** do any serialization around calls to the file system. This makes it far less likely that we will end up with a deadlock – the original intent – but it also means that we can never assume that the file won't change sizes while we are operating on it.

Of course, we have to assume file sizes can change at any time when accessing a file across the network. Simultaneous shared access to a file over the network is a reality that we must support (and will be a topic of future articles). Thus, we are prepared to deal with errors being returned to us when we issue I/O operations. But for the non-network case, how could this happen? We've observed several interesting cases:

- The file size changes while we are attempting to do I/O. This is the harsh reality of performing I/O operations without holding locks.
- The VM system may perform I/O operations while we are performing I/O. While in our own isolation filter we avoid cached writes, the file system beneath us is always free to convert a non-cached I/O into a cached I/O. We have observed this in practice with NTFS in recent versions of Windows.
- A file may discard I/O operations because it is in the process of being deleted. This makes sense, as its inefficient to do I/O to a file that's going to be discarded, but it leads to the peculiar situation of getting STATUS_SUCCESS back, even though zero bytes are written to the file. In our case, we just treat it as expected behavior and continue on ahead.

In our own work, we've found that managing file sizes was more complex than originally anticipated. While we are now successfully managing this, it took many weeks of concentrated effort and work as we constructed a viable model for how to manage this. Hopefully this brief description will help future developers in this area deal with the surprises that are in store for them.

Our Top Five for WDF Driver Devs...

(CONTINUED FROM PAGE 9)

```
kd> !wdfdriverinfo nothing_kmdf.sys 1
-----------------------------------
Default driver image name: nothing_kmdf
WDF library image name: Wdf01000
 FxDriverGlobals  0xffffe001ed4e19c0
 WdfBindInfo      0xfffff800b0883040
   Version        v1.11 build(0000)
-----------------------------------
Count of WDF Objects
-----------------------------------
WDFDRIVER = 1
WDFDEVICE = 1
WDFQUEUE = 1
WDFCHILDLIST = 1
WDFCMRESLIST = 2
WDFIOTARGET = 1
-----------------------------------

WDF Verifier settings for nothing_kmdf.sys is ON
  Pool tracking is ON
  Handle verification is ON
  IO verification is ON
  Lock verification is ON
  Device state verification is OFF
  Handle reference tracking is OFF
-----------------------------------
```

Figure 1 -- Using !WDFDRIVERINFO to show WDF Verifier options

Hint 3: Whenever You Encounter a Driver Problem, Examine the WDF Log

Got an error status back from your call to WdfIoQueueCreate? Perhaps your driver was running great, and then it suddenly crashed. In debugging a WDF driver problem, your first step should always be to examine the WDF Log.

The WDF Log is always enabled. You examine the log using the command:

!WDFLOGDUMP <drivername.sys>

Because you've enabled both WDF Verifier and Verbose Logging (as a result of reading and following Hint 1), your log can contain all sorts of additional interesting information whenever you encounter a problem. Sure, there can be a lot of nonsense about internal plug and play and power states. But, trust me, you'll get used to seeing that after a while, and you might even find it helpful someday. Until you do, just look for log entry information that's really obviously useful. When you have Verbose Logging

(CONTINUED ON PAGE 18)

Our Top Five for WDF Driver Devs...

(CONTINUED FROM PAGE 17)
enabled, I bet you'll be surprised at the very "English-Language like" information that can appear in this log to help you diagnose a problem.

Figure 2 shows an example of output from the WDF Log displayed after an error was encountered in the driver's *EvtDriverDeviceAdd* Event Processing Callback. The dev creating this example cut and pasted some code to create a WDFQUEUE, but changed the Queue Dispatch Type to Manual. You can see the results at log records 12 through 14.

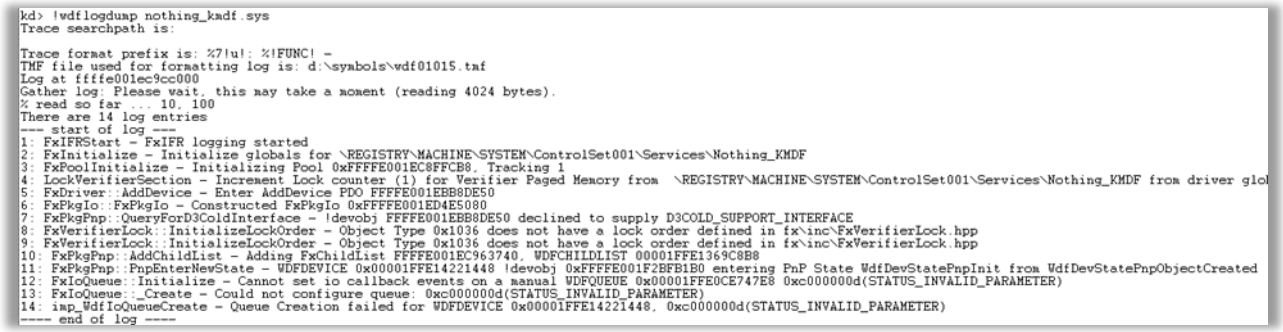


Figure 2 -- Gee... I wonder why my call to WdfIoQueueCreate failed?

Not every message is this clear, but (when verbose logging is enabled) many are. This is why it makes sense to check the log every time you find an error.

Hint 4: Those Weird NTSTATUS Values are WDF-Specific Statuses

A very common question that we get in our WDF seminar is, "What's going on! The NTSTATUS value I got back from calling a KMDF function isn't listed in NTSTATUS.H!" Correct! If you get a status value that starts with 0xC020xxxx, it's a custom WDF status value. Check the file WDFSTATUS.H in the WDK's \Include\wdf\kmdf\<version> or \include\wdf\umdf\<version> directory. There you'll find the definitions of the WDF-specific status values, some of which are very specific and helpful. Awesome, right?!

Hint 4: Don't Worry if the WDF Log Contains Odd Function Names

This is a problem that we don't see as much with new versions of KMDF than we used to. But if you're using an older version of WDF you may still run into it, so it's worth mentioning.

Let's say, for example, you call **WdfWorkItemCreate** and forget that specifying a **WDF_OBJECT_ATTRIBUTES** structure is required. You get back an error, but when you look in the WDF Log for further information you see a message that says something like:

> imp_WdfDpcCreate - WDF_OBJECT_ATTRIBUTES required, status 0xc0200212

We previously explained how to make sense out of the status, but what's with the "**WdfDpcCreate**"?? Your driver might not even call **WdfDpcCreate**.

In older versions of KMDF, the function name shown in the Log isn't always correct - So, you should just ignore it. We whined to the WDF team about it, and they've fixed most of these references in newer version of KMDF. In a recent version, for example, instead of the above message you'll get:

> FxValidateObjectAttributesForParentHandle - WDF_OBJECT_ATTRIBUTES required, NTSTATUS=C0200212

That's much better, don't you think? Thanks WDF team! Meanwhile, if you're using an older version of KMDF, don't be confused if you see the name of a function you didn't call. Just ignore the name, and pay attention to the message.

Hint 5: Use the WDF Debugger Extensions!

Our final hint is really the gateway to WDF debugging power. I used to sit across the hall from a guy who worked pretty frequently

(CONTINUED ON PAGE 19)

Our Top Five for WDF Driver Devs...

(CONTINUED FROM PAGE 18)

on WDF drivers. I'd hear him in his office cursing, trying to figure out what was going wrong in his KMDF code. I think he felt like he never really knew what was happening, between the Event Processing Callbacks, the WDF-managed state, and the opaque WDF Objects. He could debug a WDM driver in a flash – but trying to run down problems in a KMDF driver often left him frustrated.

The hint he was missing was that he needed to make better use of the WDF Kernel Debugger Extensions. These extensions are **the key** to WDF driver debugging happiness. When I'm debugging a WDF driver, I usually like to start by viewing the driver's WDF Object hierarchy, and the handles associated with each of the objects. This is easily done by using the !WDFDRIVERINFO command with a flags value of 0x70 as shown in *Figure 3*.

```
kd> !wdfdriverinfo nothing_kmdf.sys 0x70
---------------------------------
Default driver image name: nothing_kmdf
WDF library image name: Wdf01000
 FxDriverGlobals  0xffffe001ec8ffc50
 WdfBindInfo      0xfffff800b1713040
   Version        v1.13 build(0000)
---------------------------------
Driver Handles:
!WDFDRIVER 0x00001ffe136dd198
    dt FxDriver 0xffffe001ec922e60
    <no associated contexts or attribute callbacks>

    !WDFDEVICE 0x00001ffe116116a8
        dt FxDevice 0xffffe001ee9ee950
        context:  dt 0xffffe001ee9eec40 NOTHING_DEVICE_CONTEXT (size is 0x4 bytes)
        <no associated attribute callbacks>

        WDF INTERNAL
            dt FxDefaultIrpHandler 0xffffe001ed2bf410
        WDF INTERNAL
            dt FxPkgGeneral 0xffffe001eea926b0
        WDF INTERNAL
            dt FxWmiIrpHandler 0xffffe001f2fdd080
        WDF INTERNAL
            dt FxPkgIo 0xffffe001f2d51550
            !WDFQUEUE 0x00001ffe0d181f78
                dt FxIoQueue 0xffffe001f2e7e080
                <no associated contexts or attribute callbacks>

        WDF INTERNAL
            dt FxPkgFdo 0xffffe001ec49b4c0
        !WDFCMRESLIST 0x00001ffe12b030c8
            dt FxCmResList 0xffffe001ed4fcf30
            <no associated contexts or attribute callbacks>

        !WDFCMRESLIST 0x00001ffe133cc8e8
            dt FxCmResList 0xffffe001ecc33710
            <no associated contexts or attribute callbacks>

        !WDFCHILDLIST 0x00001ffe136bd198
            dt FxChildList 0xffffe001ec942e60
            <no associated contexts or attribute callbacks>

        !WDFIOTARGET 0x00001ffe133cb458
            dt FxIoTarget 0xffffe001ecc34ba0
            <no associated contexts or attribute callbacks>

---------------------------------
```

Figure 3 - WDF Object Hierarchy

In *Figure 3*, you can see we have a WDFDRIVER Object (with no context) that has a single WDFDEVICE associated with it. That context is of type NOTHING_DEVICE_CONTEXT, and if we want to see what's in the device context area, we can dump it using the DT command shown.

It's a bit hidden (look carefully in *Figure 3* to find it, buried after the 4th WDF INTERNAL object in the handle list), but you can also see that there's a single WDFQUEUE associated with the WDFDEVICE. You can just cut and paste the WDFQUEUE command shown in the output to find information about the Queue. This is shown in *Figure 4 (page 20)*.

(CONTINUED ON PAGE 20)

Our Top Five for WDF Driver Devs...

(CONTINUED FROM PAGE 19)

```
kd> !WDFQUEUE 0x00001ffe0d181f78

Dumping WDFQUEUE 0x00001ffe0d181f78
=======================
Sequential, Default, Not power-managed, PowerOn, Can accept, Can dispatch, ExecutionLevelDispatch, SynchronizationScopeNone
    Number of driver owned requests: 0
    Number of waiting requests: 0


    EvtIoRead: (0xfffff800b1711230) Nothing_KMDF!NothingEvtDeviceAdd
    EvtIoWrite: (0xfffff800b1711280) Nothing_KMDF!NothingEvtDeviceControl
    EvtIoDeviceControl: (0xfffff800b17111f0) Nothing_KMDF!NothingEvtDeviceAdd
```

Figure 4 -- WDFQUEUE, its attributes and Callbacks

In the output shown in *Figure 4*, you can see the Queue's attributes. Note that a PASSIVE_LEVEL Execution Level constraint has not been applied ("ExecutionLevelDispatch" is shown) and there's no Sync Scope associated with this Queue ("SynchronizationScopeNone").

You can also see that the Queue is empty ("Number of waiting requests" is zero) and there are no Requests in progress ("Number of driver owned requests" is zero). This Queue handles Read, Write, and Device Control functions via the Event Processing Callbacks shown. Note that there's no Event Processing Callback provided for Internal Device Controls, so this driver does not handle that type of I/O operation. With the names of the I/O Event Processing Callbacks, you can easily cut and paste to set breakpoints on any of these routines if you desire.

There's also a huge amount of information you get about your device, its capabilities, and its state using the !WDFDEVICE command shown in *Figure 3*. Just by cutting and pasting that command (and adding the flags 0xFF, which means "give me everything!") you get the large collection of information shown in *Figure 5 (page 21)*.

The output in *Figure 5* is pretty much self-explanatory. Note that this can help you diagnose both PnP and Power issues. You also have the WDM Device Object pointers that correspond to the WDFDEVICE, its upper filter, and its PDO. So, you quite easily can take your debugging into the realm of WDM if necessary.

These commands are only examples. There are commands that are equally powerful for just about every WDF Object. And what's **really** cool is that you can use just about all of these commands in UMDF V2, just like you can in KMDF.

There You Have It

So there are our top five hints for making your life developing and debugging WDF drivers easier. We'd like to hear your hints. If

(CONTINUED ON PAGE 21)

Our Top Five for WDF Driver Devs...

(CONTINUED FROM PAGE 20)

```
kd> !WDFDEVICE 0x00001ffe116116a8 0xFF

Dumping WDFDEVICE 0x00001ffe116116a8
================================

WDM PDEVICE_OBJECTs:  self ffffe001ec403520, attached ffffe001ebb8de50, pdo ffffe001ebb8de50

Pnp state:  119 ( WdfDevStatePnpStarted )
Power state:  307 ( WdfDevStatePowerD0 )
Power Pol state:  565 ( WdfDevStatePwrPolStarted )

Default WDFIOTARGET: 00001ffe133cb458

No pended pnp or power irps
Device is the power policy owner for the stack

Pnp state history:
[0] WdfDevStatePnpObjectCreated (0x100)
[1] WdfDevStatePnpInit (0x105)
[2] WdfDevStatePnpInitStarting (0x106)
[3] WdfDevStatePnpHardwareAvailable (0x108)
[4] WdfDevStatePnpEnableInterfaces (0x109)
[5] WdfDevStatePnpStarted (0x119)

Power state history:
[0] WdfDevStatePowerObjectCreated (0x300)
[1] WdfDevStatePowerStartingCheckDeviceType (0x316)
[2] WdfDevStatePowerD0Starting (0x30f)
[3] WdfDevStatePowerD0StartingConnectInterrupt (0x310)
[4] WdfDevStatePowerD0StartingDmaEnable (0x311)
[5] WdfDevStatePowerD0StartingStartSelfManagedIo (0x312)
[6] WdfDevStatePowerDecideD0State (0x313)
[7] WdfDevStatePowerD0 (0x307)

Power policy state history:
[0] WdfDevStatePwrPolObjectCreated (0x500)
[1] WdfDevStatePwrPolStarting (0x501)
[2] WdfDevStatePwrPolStartingPoweredUp (0x583)
[3] WdfDevStatePwrPolStartingSucceeded (0x502)
[4] WdfDevStatePwrPolStartingDecideS0Wake (0x504)
[5] WdfDevStatePwrPolStarted (0x565)

Idle state history:
[0] FxIdleStarted (0x2)
[1] FxIdleStartedPowerUp (0x3)
[2] FxIdleDisabled (0x5)
[3] FxIdleDisabled (0x5)

Power references: 0

S0Idle policy not configured

SxWake policy not configured

Power Capabilities:
  DeviceWake: PowerDeviceUnspecified
  SystemWake: PowerSystemUnspecified
  S-D mapping:
    S0->PowerDeviceD0
    S1->PowerDeviceD3
    S2->PowerDeviceD3
    S3->PowerDeviceD3
    S4->PowerDeviceD3
    S5->PowerDeviceD3
  IdealDxStateForSx: PowerDeviceD3
```

Figure 5 - Everything you always wanted to know about your WDFDEVICE

you've got a hint to share, tweet it to us @OSRDrivers . We'll find a prize lying around the office for the best hint that's tweeted to us in the two weeks following the release of this issue of *The NT Insider*. C'mon… let's hear **your** hints!

Follow us!

Xperf… (Cont.)

(CONTINUED FROM PAGE 7)

We then start the driver, let it run for a while, and dump out our context structure in the debugger. Much to our surprise, on a seemingly idle system we've had a good number of misses:

```
0: kd> dt 0xffffe001e984c600 Nothing_KMDF!NOTHING_DEVICE_CONTEXT
   +0x000 Nothing          : 0
   +0x008 Timer            : 0x00001ffe`166eda48 WDFTIMER__
   +0x010 LastTimestamp    : _LARGE_INTEGER 0x01d0511c`90e0118a
   +0x018 Hits             : 0x13a8
   +0x01c Misses           : 0x43
```

0x43 injured people in a few seconds seems like a bad situation for a product, so we want to investigate more. What happened during those 0x43 occurrences that led to the missed timer period??

Enter Xperf and the Windows Performance Analyzer

This scenario is exactly what Xperf is all about. Xperf gives us the ability to peer into the state of the Windows OS and see **exactly** what was happening over a collection period. Xperf is provided as part of the Windows Performance Toolkit, which installs along with the WDK.

Xperf itself is simply a utility to enable and collect data from Event Tracing for Windows (ETW) Providers. There is one provider, the **NT Kernel Logger**, which provides kernel level trace data from the OS and any Microsoft supplied drivers (e.g. WDF, Filter Manager, etc.). Additionally, there are any number of "User Mode Providers", which despite the name, can exist either in user mode or third party kernel mode. User Mode Providers are identified and enabled either by GUID or by a registered name.

To start our analysis, we're going to focus on enabling the **NT Kernel Logger**. There are various flags that we can enabled for this provider and we can see them by running: **xperf.exe –providers kf**. For this trace, we're going to choose to enable the following flags in the **NT Kernel Logger**:

- PROC_THREAD – Records process and thread activity
- LOADER – Records the loaded module list from the system, which will allow us to resolve function addresses into function names using PDBs
- DPC – Deferred Procedure Call activity. WDFTIMERs are really just DPCs under the hood, so this is an important one
- INTERRUPT – Interrupt Service Routine activity. Interrupts can interrupt DPCs, so interrupts might be the reason for our misses
- WDF_DPC – We're using a KMDF driver for this activity, so this is an important flag to set. Using this flag, the **NT Kernel Logger** will work in conjunction with WDF to record **our** DPC routine information instead of the Framework's. Without this flag, all DPCs used by all KMDF drivers would be bundled together in the trace (i.e. every DPC event logged from a KMDF based driver would point to the one Framework DPC wrapper)
- WDF_INTERRUPT – Same as WDF_DPC, except for Interrupt Service Routines. We don't have an ISR in this driver, so this one is just for completeness

We enable all of the above with the command shown in *Figure 2*:

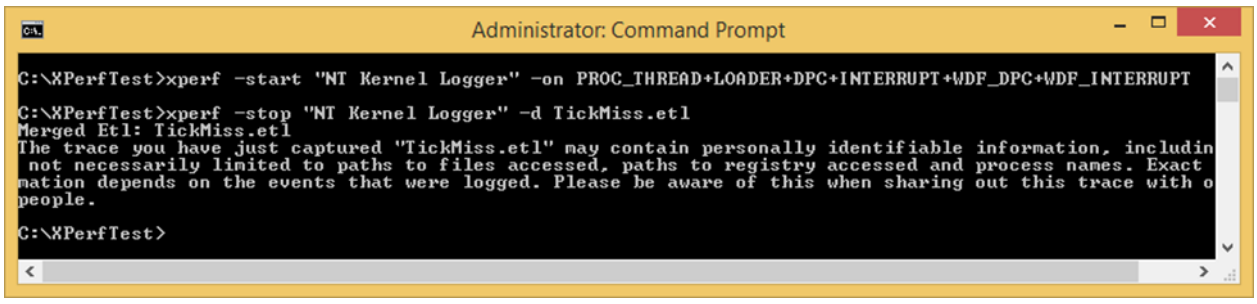


Figure 2 — Starting (and then Stopping) an NT Kernel Logger Trace

(CONTINUED ON PAGE 23)

Xperf… (Cont.)

(CONTINUED FROM PAGE 22)

We can now enable our driver, allow a few seconds to pass to collect some data, and then stop the trace. The resulting log data will be collected to an Event Trace Log (ETL) file that we specify. In this case, we'll store the results into the *TickMiss.etl* file (again, see *Figure 2*).

Now for the fun part. By double clicking the resulting ETL file, we will open the trace data in the Windows Performance Analyzer (WPA) application (*Figure 3*).

Figure 3 — WPA Default View

On the left hand side of WPA, we see entries for each of the ETW Providers and flags that we enabled. Before proceeding any further, we want to translate any function addresses in the trace into symbolic names. We'll do this by clicking Trace->Load Symbols.

Once the symbol files have loaded, we want to dig in to some details about what was happening on the system. The view of the data that we're going to focus on for this article is the **DPC/ISR Timeline by Module, Function**, which you can find under the

(CONTINUED ON PAGE 24)

DID YOU KNOW?

We use and teach with the most recent released version of Windows in all our public seminars. This makes it easy to learn the latest advancements in the O/S and tools. But hey, we've been Windows kernel devs since the first release of NT, and are happy to walk down memory lane on a particular topic if you're company or customers aren't so forward-looking.

Xperf… (Cont.)

(CONTINUED FROM PAGE 23)

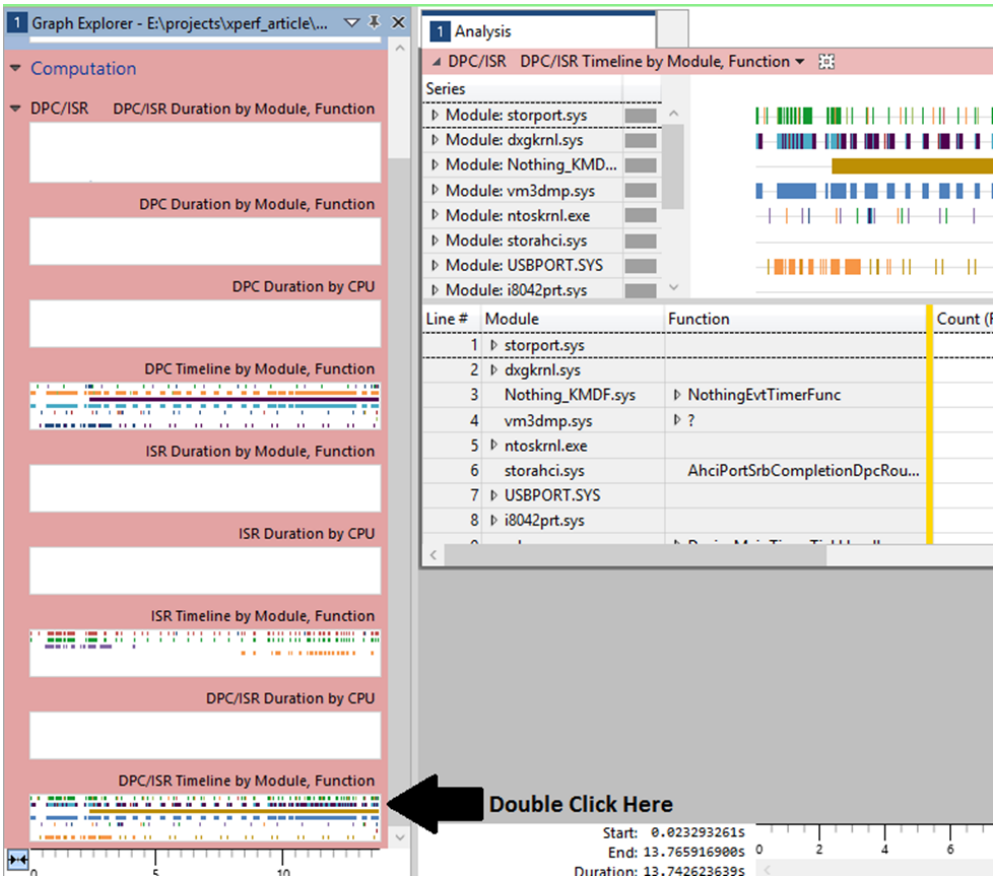


Figure 4 — Analysis View of DPC/ISR Timeline

Computation entry in the Graph Explorer. Once you have located this view, double click on it to bring it into the Analysis window. The resulting display will look something like *Figure 4*.

What we see in the resulting Analysis window is a timeline showing the duration of every DPC and every ISR that happened during the trace. The DPCs and ISRs are grouped by the module that they occurred in and we do indeed see invocations to our *NothingEvtTimerFunc* happening in the Nothing_KMDF.sys module. By clicking on this entry in the display, WPA will highlight each execution of the callback in the timeline, as shown in *Figure 5*.

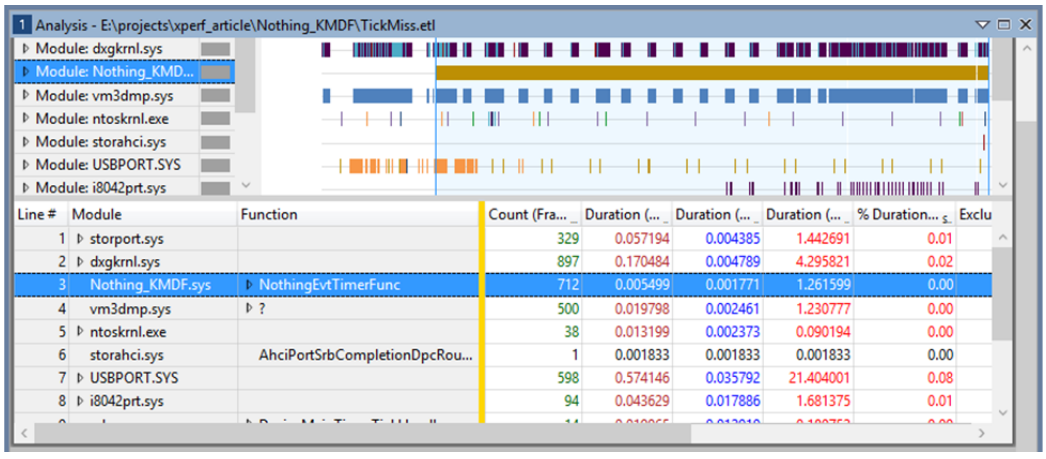


Figure 5 — Our Timer Callback Invocations

(CONTINUED ON PAGE 25)

Xperf... (Cont.)

(CONTINUED FROM PAGE 24)

If we right click at this point and select Zoom, the display will zoom in and only show the period of the timeline where there are events from our driver. This allows us to immediately exclude anything in the trace that happened before our driver was loaded, which is a useful first pass at isolating what interests us.

Additionally, we can use the mouse to highlight a custom range and then zoom into that, which you can see in the following two steps. First we highlight, as shown in *Figure 6*.

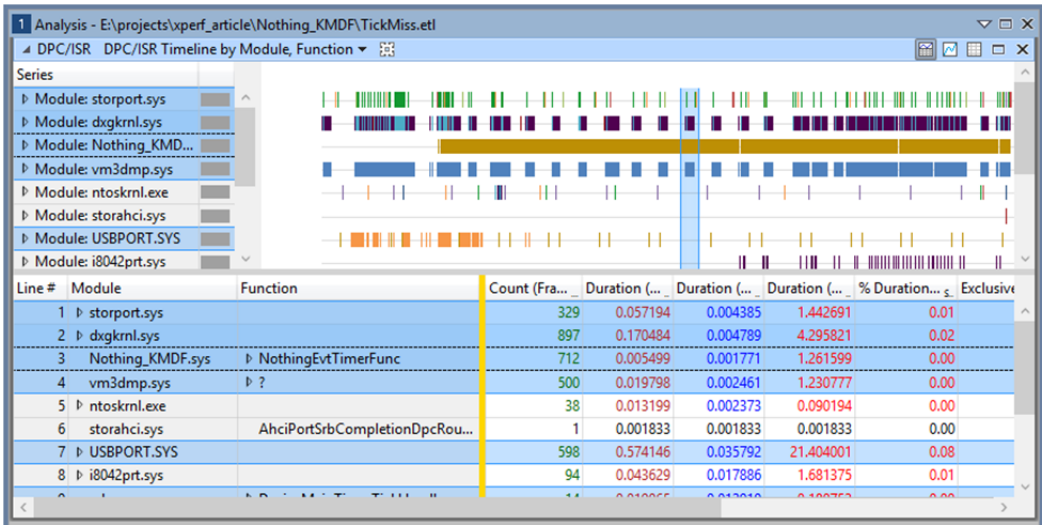


Figure 6 — Highlighting a Custom Page

And then we zoom by Right Clicking and selecting Zoom. The result of the zoom is shown in *Figure 7*.

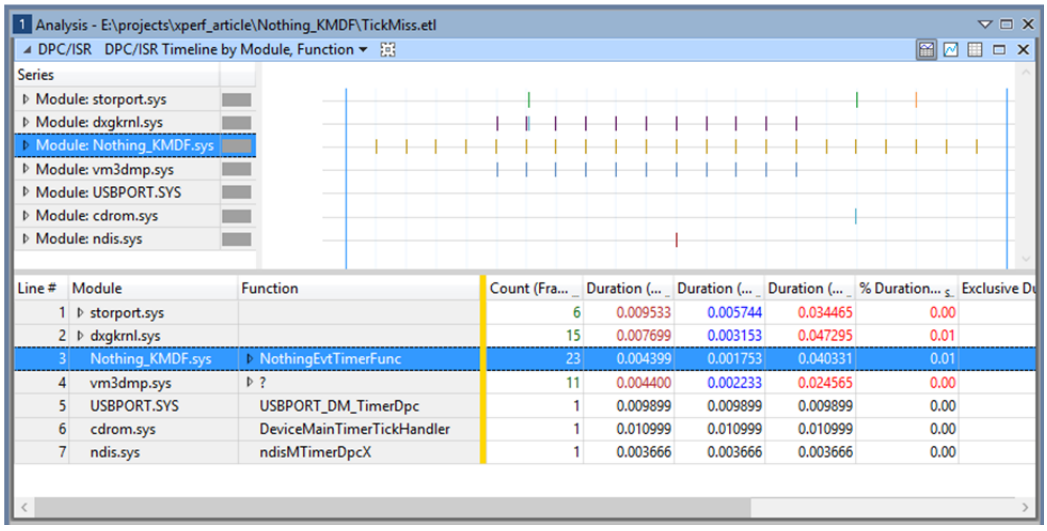


Figure 7 — Zooming in to a Custom Range

Pretty awesome, right? We get to see exactly which DPCs and ISRs were running during this time period and potentially use that to locate where our delays are coming from.

(CONTINUED ON PAGE 26)

Xperf… (Cont.)

(CONTINUED FROM PAGE 25)

Needle in a Haystack?

Of course, the astute reader is probably thinking "Great. I am now full of awe and am heading off to do volunteer snow shoveling of roofs in New England. However, when I get back I have no idea **which** of these events corresponds to a missed timing."

The trouble of course is that this trace is giving us **everything**, not just the moments of time where we get a miss of our timing window. According to *Figure 5*, there were 712 unique invocations of our timer callback in this trace (the **Count** column). Some small percentage of these indicate misses, so how are we supposed to isolate the "bad" invocations without going through all 712?

This is where adding our own, custom trace points to the Xperf analysis comes in. This involves turning our driver into an ETW Provider and generating trace events at the appropriate time. Once our driver has trace events, we simply need to enable our ETW Provider along with the **NT Kernel Logger**. This way, we'll have both sets of trace events in the resulting ETL and can use our custom trace events to focus our analysis.

Creating an ETW Provider

Turning our driver into an ETW Provider is a simple process, even if the syntax might be a bit confusing. Here are the steps we need to follow:

1. Create an "instrumentation manifest" that describes our trace events. This is just an XML file that we can either create by hand or by using the SDK's Manifest Generator utility (EcManGen.exe)
2. Modify our project to invoke the Message Compiler to turn our manifest into:
    a. A binary resource blob to include in our image
    b. A header file with generated functions to register our ETW Provider, unregister our ETW Provider, and generate trace events
3. Include the header file generated in Step 2
4. Add trace events to our code

The manifest file is pretty ugly, but there is much documentation and many samples for both user mode and kernel mode available. The rest of the steps are the same in user mode and kernel mode and are relatively straightforward.

In our example, the important part of our instrumentation manifest is the portion that describes our one event that we log (full manifest is provided with the sample code). This event will be logged any time we experience a miss in our timing window and will include with it the time delta:

```
<templates>
  <template tid="tid_Expiration">
    <data
        inType="win:UInt64"
        name="Delta"
        outType="win:HexInt64"
        />
  </template>
</templates>
<events>
  <event
      channel="SYSTEM"
      level="win:Informational"
      message="$(string.ExpirationMiss.EventMessage)"
      opcode="win:Info"
      symbol="ExpirationMiss"
      template="tid_Expiration"
      value="1"
      />
</events>
```

The **template** describes what data we'll be logging with our trace event. In this case, we're going to input a single 64-bit value named "Delta" and we want that output in Xperf as a 64-bit hex value.

(CONTINUED ON PAGE 27)

Xperf… (Cont.)

(CONTINUED FROM PAGE 26)

The **event** describes the actual event, which refers to the **template** to supply input and output data.

Once we have compiled this manifest and included the appropriate header in our project, we can call a generated function named **EventWriteExpirationMiss** (this is "**EventWrite"** followed by the symbol tag from the above manifest). We'll provide to this function an optional activity ID (for matching up related trace events) and the delta from our calculation:

```
if (delta > ((20 * 1000 * 1000) / 100)) {
    devContext->Misses++;
    EventWriteExpirationMiss(NULL, delta);
} else {
    devContext->Hits++;
}
```

Registering Our Provider

On our test system, we'll want to register our manifest as an ETW Provider. This makes starting and stopping traces easier, as we can do it by name instead of GUID, and allows Xperf to put some additional information in the ETL file. Open an elevated command prompt and import the manifest using the **wevtutil.exe** utility as shown in *Figure 8*.

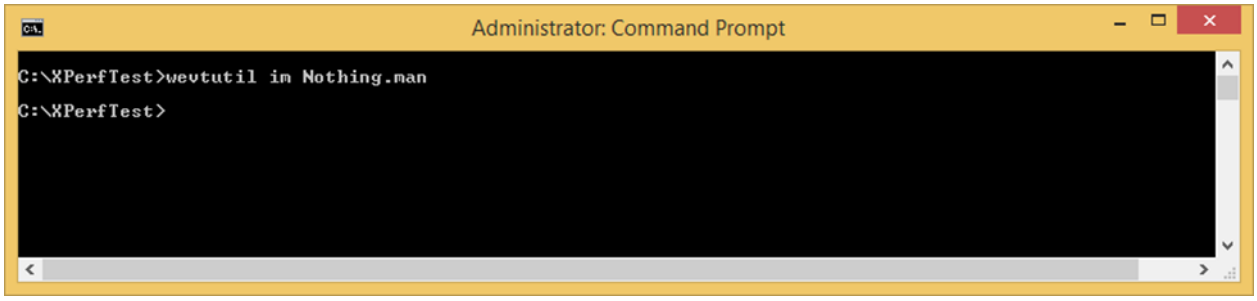


Figure 8 — Importing a Manifest

(CONTINUED ON PAGE 28)

Xperf… (Cont.)

(CONTINUED FROM PAGE 27)

Starting a New Trace

We now need a new Xperf trace that includes our custom trace data. We'll do that by enabling both the **NT Kernel Logger** as well as our custom provider on the **xperf.exe** command line. This is as easy as adding another **–start** directive, followed by a custom name for this trace, followed by the registered name in the manifest (in this case "OSR Nothing KMDF"). The full command line is shown in *Figure 9*.

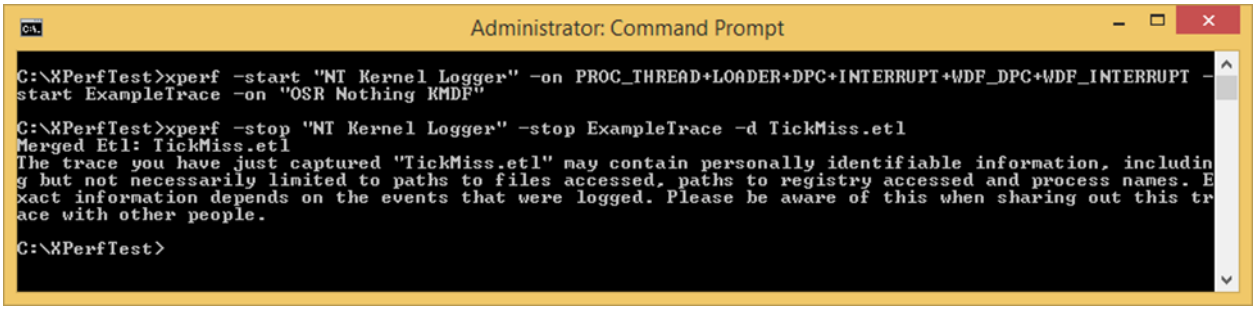


Figure 9 — Starting (and then Stopping) Multiple ETW Providers

After a period of time, we'll then want to stop both ETW Providers. Again, we simply need to specify two **–stop** directives on the command line (again, *Figure 9*). Trace data from both providers will appear in the resulting ETL file.

Finding Our Custom Trace Events

It's not intuitively obvious where our trace events can be found in the WPA, but once you know where they are it sort of makes sense. Because we're a third party driver generating trace data, we are considered to be a "User Mode Provider" (yes, even though we traced this data in kernel mode). User Mode Providers are found in WPA under **System Activity->Generic Events**. The events here are listed in a timeline and grouped by the ETW Provider. Thus, double clicking on this entry in the Graph Explorer shows us our custom trace events in the Analysis view (*Figure 10*) along with any other User Mode Providers that were enabled.

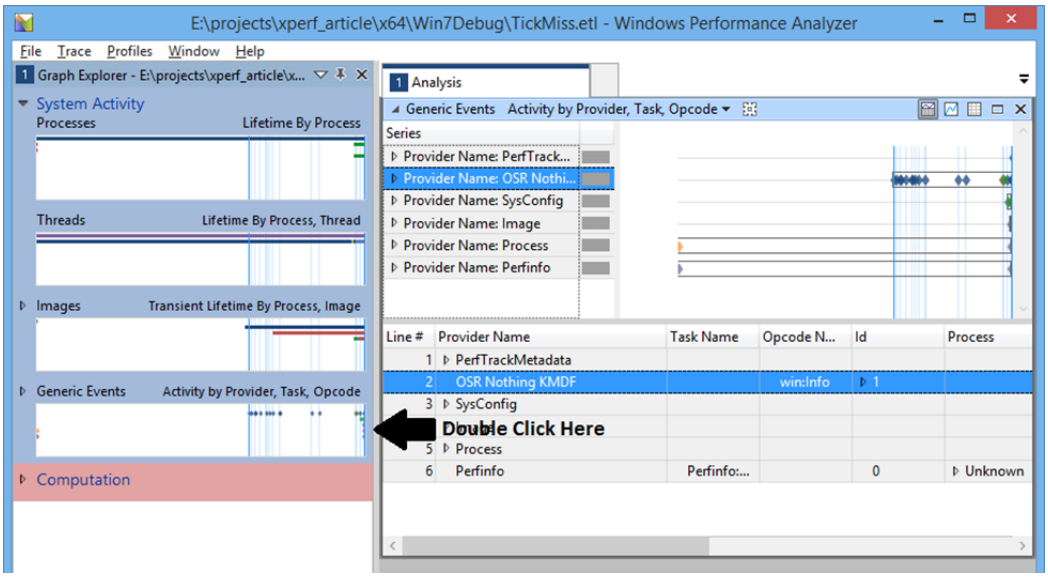


Figure 10 — Custom ETW Provider Data

Using this, we now know which parts of the timeline include misses of our deadline. We can use the zoom in feature of WPA to zoom in to a particular time period, then explore the other ETW Provider data within that period to determine the cause.

(CONTINUED ON PAGE 29)

Xperf… (Cont.)

(CONTINUED FROM PAGE 28)

As an example, *Figure 11* zooms in to the period in between two invocations of the timer callback. Each of those invocations is shown by a blue line on the timeline. The first invocation hit the deadline, but the next invocation did not (as evidenced by the existence of our custom trace event, shown as a diamond on the "Provider Name: OSR Nothing_KMDF" series in the top right)). By zooming all the way in to this time period, we can see that in between these two invocations there was a DPC or ISR from the graphics subsystem (dxgkrnl), VMware (vm3dmp.sys), and the OS itself (ntoskrnl). In addition, the storage controller (storport)

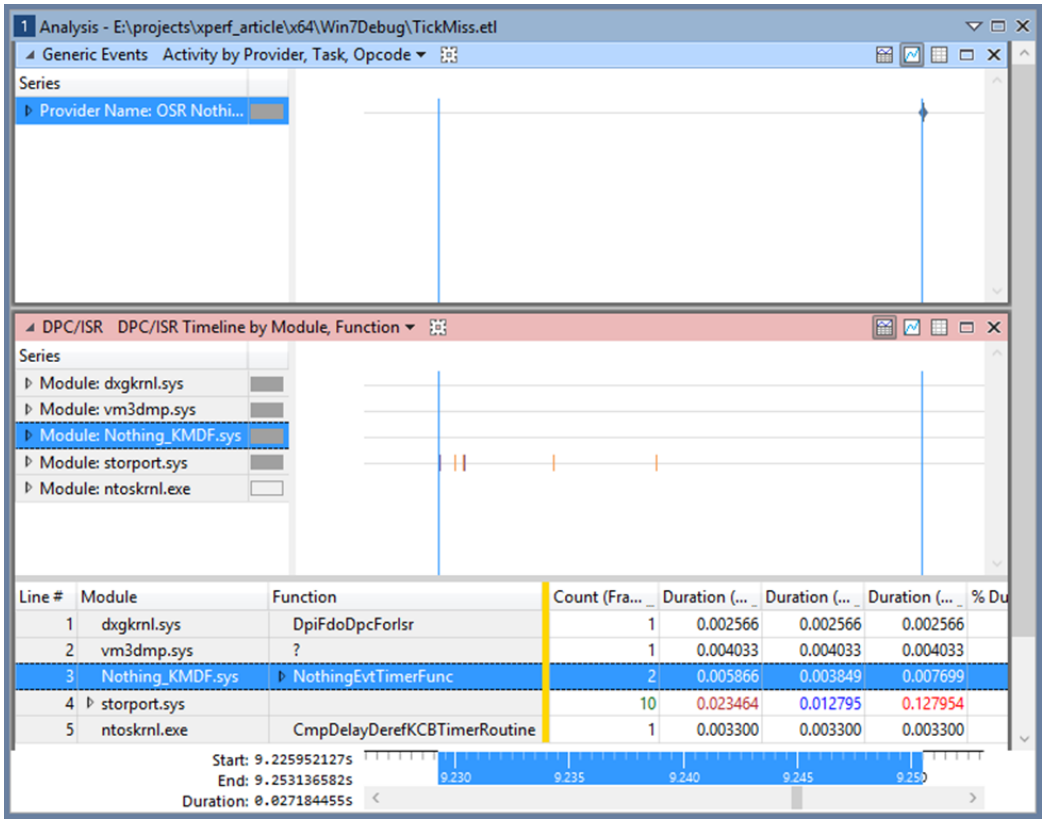


Figure 11—Period with Missed Deadline

(CONTINUED ON PAGE 30)

Xperf… (Cont.)

(CONTINUED FROM PAGE 29)
generated 10 DPCs or ISRs in this window. Even though you can't see each of these on the DPC/ISR Timeline in Figure 11 (they **are** there, you just would have to zoom in a bit further to see 1 instance of an event), you can see these events listed in the table at the bottom.

What we have here, then, is clear evidence that there was nothing wrong with the system, it was just busy, and that Windows is not suited to this sort of task. However, using Xperf we don't need to "guess" that this is true, we can actually collect the data and see for ourselves.

Try It Yourself!

That's a lot of screen grabs and bits of information and we barely scratched the surface, but hopefully you're awestruck enough to want to try it yourself. When it comes to Xperf practice is **truly** the best way to learn, as it takes exposure and perseverance to be able to interpret any of this data. That's why we're providing the source to our example along with this article. Install it, run the experiments, and start being a happier person today!

Code associated with this article:
http://insider.osr.com/2015/code/xperf.zip

Follow us!

Peter Pontificates... (Cont.)

(CONTINUED FROM PAGE 5)

So, the way things look today, on Windows 10 you get your itty-bitty, Windows 95 and Windows 7 style, list of folders back. And while you can increase the size of the start menu to full screen, the useful Windows 8.1 style category view for desktop apps is nowhere to be found.

Figure 4—Windows 10 Start Menu (image courtesy BetaNews.com)

I call this change. But I do not call this progress. Progress means moving forward. Towards "betterment." Going back to an interface designed years ago for little screens is not better. It's silly.

There's a saying in English "The customer is always right" – In French they say the analogous "le client n'a jamais tor." I say this is complete bullshit. In tech, the customer is almost never right. Because the customer almost always wants exactly what they're used to, what they've had for years, until they try, **seriously try**, the new thing you're giving them. And when it comes to user interfaces "change aversion" is legendary. The problem is with the Internet echo-chamber, it's easier to just repeat the latest meme than spend the time to think through an issue.

So, the Windows 95 / Windows 7 start menu is definitely not superior to the Windows 8.1 start screen. Not by any rational standard I can think of. And returning to a 20 year old design in Windows 10 does not seem like a good way to make progress to me.

But, I think the real question is why does it have to be one way or the other? I think the **only** mistake Microsoft made in Windows 8 was **forcing** the start screen on people, and not providing the older start menu interface as an option during the transition period. Make it so people can use the old way that they've come to know and love, while they have the opportunity to eventually try out the newer interface. If they had done this, I bet the reaction to the changes in Windows 8 wouldn't have been even half as vehement.

And so, for Windows 10… if you **have** to give us the little bitty strip of application folders, can we at least **also** have access to the Windows 8.1 start screen somehow? Maybe when you enlarge the Win 10 start menu to full screen, it becomes the older Windows 8 apps start screen, the one you see when you select the down arrow?

Well, there's plenty of time for the look and feel of Windows 10 to change. No date has yet been announced for its release. And in the time that's left, I hope Team Nadella gives the rabble the start menu they want today, **and** the start screen they'll come to appreciate if they ever freaking try it.

If they don't give us a start screen, maybe we can raise a fuss on the Internet. Wouldn't that be funny. People clamoring for the return of the Windows 8.1 start screen.

Nah. It won't be an Internet meme. Never happen.

Follow us!

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

OSR Seminar Schedule

Seminar	Dates	Location
Developing File Systems	12-15 May	Boston/Waltham, MA
Internals & Software Drivers	18-22 May	Dulles/Sterling, VA
WDF Drivers: Core Concepts	8-12 June	Boston/Waltham, MA
WDF Drivers: Advanced Implementation Techniques	15-18 June	Boston/Waltham, MA
Kernel Debugging & Crash Analysis	3-7 August	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've learned everything that I wanted to learn and quite a bit that I did not know I needed.
- I think Scott nicely catered to a diverse audience, some of whom had only taken intro to OS and some who already had experience developing drivers.
- Scott was fantastic. He was very helpful at trying to meet each student's needs. **I will highly recommend him and OSR to my associates.**
- "Peter's **style of teaching is excellent** with the kind of humor and use of objects to give you a visual representation of how things work that was simply amazing.
- "I was very nervous coming in to the seminar as I wasn't sure I had enough hands on experience and background knowledge on Windows internals. The class put my mind at ease and **I was able to quickly grasp and understand the concepts**."
- "I was pleased with the experience. As someone new to driver development, it gave me a better understanding of the framework and **made me a lot more comfortable working with our driver code**."

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

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

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.