ISSUE VOLUME
03/20
SEPT-OCT
2014

The NT Insider

A publication of OSR Open Systems Resources, Inc.

```
Scope(\_SB)
{
    //
    // Declare NT Insider
    //
    Device(NTIN)
    {
    Name(_HID, "Iss3Vol20")

    Name(_ADR, Zero)
    Name(_UID, One)

    Name(_DDN, "The NT Insider Sept/Oct 2014")

    Name(_DEP, Package(0x4)
    {
        WDK,
        WDM,
        WDF,
        FSF         // File system filters
    })

    Method(_STA, 0x0, NotSerialized)
    {
        Return(0xf)
    }

    Method(_CRS, 0x0, Serialized)
    {
        Name(RBUF, ResourceTemplate()
        {
            //
            // Peter Pontificates
            //
            Pontification("To the Cloud?", 0x4)

            //
            // Serialization is important.  But does KMDF's
            // Sync Scope feature help?  How exactly does this work?
            //
            StaffArticle("Understanding Sync Scope in WDF Drivers", 0x6)

            //
            // Never worry about copying your executables to your
            // target when debugging again!
            //
            StaffArticle("Updating Drivers with KDFiles", 0x8)

            //
            // Expert Daniel Terhell discusses some of the challenges
            // of dealing with processing latency in Windows drivers.
            //
            GuestArticle("Windows and Real-Time", 0x10, "Daniel Terhell")

           //
            // A real, working, model of a file system filter
            // using the Isolation Filter model.
            //
            StaffArticle("Isolation Realized", 0x12)

        })
        Return(RBUF)
    }
}
```

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

OSR Seminars Return to ASIA in 2015

Tell Us: **Where** Do You Want Us to Teach?

It's been a while since we've scheduled our seminars in Asia. But we've heard your requests for more training. Therefore, we're planning to send members of the OSR team on a "road trip" to various countries in Asia in 2015.

When?
Current plan is late January and early February 2015.

Where?
Tell us where you would like us to schedule our next OSR public seminars. Our first priority will be locations in India, but we will also consider other locations including China, Korea, Japan, and Singapore. Basically, we'll come to wherever a Public OSR seminar is needed and there are enough people to make the trip worthwhile.

Note: Private seminars for your company can also be arranged.

What?
Special versions of our two most popular seminars: Writing WDF Drivers: Core Concepts and Kernel Debugging and Crash Analysis. We will also consider offering other seminars, if there is sufficient demand.

You Make It Happen!
If you're interested in attending, helping to organize, or even partnering with us to bring a public OSR seminar to your region… we want to hear from you!

This is also a great opportunity to schedule a customized, private, OSR seminar for your company at special reduced rates.

Email the OSR seminar team at seminars@osr.com to let us know your interest in bringing OSR Seminars to Asia. Maybe you're just one guy who would like to attend the training: email us! Maybe you're a manager at a company who would like to organize a seminar for his team: email us! Or perhaps you're interested in helping to organize a public seminar in your region: email us! We will respond to your inquiry quickly and provide additional information about our seminar plans.

We're looking forward to hearing from you, and making our "road trip" as helpful as it can be!

Coming in 2015:

书写 WDF 驱动程序

WDF 드라이버 작성

Get Social with OSR
Real -Time Updates

Suffice it to say that if you're reading these updates in this issue, you're not getting the real-time aspect of OSR's social media presence. Follow us on OSR's Dev Blog, Twitter, Facebook , LinkedIn, or via the new OSRHINTS list (see below for how to join) to be "in the know" with the rest of our followers!

A Second Chance for WPP Tracing (video + article)
Believe it or not… OSR is giving WPP Tracing another chance. Scott explains what's up:
http://www.osr.com/blog/2014/08/26/second-chance-wpp-tracing/

Follow us!

UMDF V2: Good or Bad Thing? (video + article)
Peter and Scott hash out their input on the topic.
http://www.osr.com/blog/2014/08/18/umdf-v2-good-bad-thing-video/

Sync Scope with Timers and Work Items
Great primer for a bigger discussion on Sync Scope in THIS issue
http://www.osr.com/blog/2014/08/07/sync-scope-timers-work-items/

When is a WHILE Not a while?
Source code should be clear, easy to understand and maintainable. Here, Peter takes issue with a specific coding practice in WDK samples that he just had to get off his chest.
http://www.osr.com/blog/2014/07/31/while-not-while/

Setting up a 1394 Kernel Debug Connection (Video)
This topic might be considered low-hanging fruit by some, but others still struggle with setting up a 1394-based kernel debugging session. Scott Noone explains all in this video.
http://www.osr.com/blog/2014/06/30/setting-1394-kernel-debug-connection-video/

Using WDF Queues with Manual Dispatch Type (Video)
Watch and listen for two different perspectives (Peter and Scott) on a particular use of WDF Queues.
http://www.osr.com/blog/2014/06/23/using-wdf-queues-manual-dispatch-type-video/

What WDK Version Am I Running?
Not as easy a question to answer as you might think.
http://www.osr.com/blog/2014/05/28/wdk-version-running/

Kx Headers in the Windows 8.1 WDK
Kx? What is the Kx prefix? Scott Noone tells all.
http://www.osr.com/blog/2014/04/18/kx-headers-windows-8-1-wdk/

EvtIoInCallerContext Callback: Called Even for I/O Operations You Don't Queue
Ah, if you could have been a fly on the wall for this discussion.
http://www.osr.com/blog/2014/04/15/evtioincallercontext-callback-called-even-io-operations-dont-queue/

Scott offers a keyboard tip on breaking into the host while debugging
http://twitter.com/OSRDrivers/status/496383324316000256

How do you dump all open File Handles in a kernel debug session?
http://twitter.com/OSRDrivers/status/491667207450292224

Come Join Us!

Writing WDF Drivers

Boston, MA
22 September

seminars@osr.com

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

Peter Pontificates:
To The Cloud?

It seems to me that we are in the midst of one of the greatest cases of mass delusion of all time. You should feel lucky, because I am apparently one of very few people who have been blessed with the ability to see this and to bring the world to its senses. And now I'm going to enlighten you.

You think the Dancing Plague of 1518 was strange? You think the mass delusion that's causing the majority of the developed world to ignore problems like resource depletion and global climate change is important and scary? These are nothing, nothing at all, compared to the trend that's been overtaking otherwise sane organizations over the past few years. That trend is moving things "to the cloud."

As I so often find myself writing in these pontifications, **I just don't get it**. Now, let me be clear: I don't think everything about the cloud is bad or wrong. The cloud certainly has its place. For example, I agree that storing the pictures of your kids on (two different) cloud storage servers can save you space locally, and make it easier to show the pics to Grandma when you visit. Heck, there's even a place in this world for Software as a Service (SaaS). Choosing the right SaaS can save you the upfront cost and annoyance of licensing, installing, and the ongoing pain of maintaining certain application that can help your business.

What makes me stand in awe, mouth agape, shaking my head in disbelief is the willingness of what I think of as well-run and level-headed organizations to store reams of their highly sensitive and confidential business data "in the cloud." I just don't see how this can make sense to anyone.

Over the past couple of years we've been upgrading the IT infrastructure here at OSR. This has included replacing all our servers and network components and implementing an entirely new backup and disaster recovery scheme. In talking with IT consultants about backups, their first recommendation was always some incredibly clever cloud-based backup scheme. The rationale? It's stored off-site in case of a disaster, you can access the backup data directly from other locations, and the amount of storage is never limited. Sounds good, right?

Maybe. At least, it sounded sort of good until I asked if the backup data was encrypted. "Oh, yes… the data is 100% safe and encrypted! In fact, everything is encrypted via 2048-bit SSL," they told me with pride. When I pointed out that this meant the data was encrypted while it was being transported on the wire, but not while it sat on the backup company's servers, they nodded eagerly: "Yup, it's fully encrypted during transit."

"What about the data at rest, on the backup company's servers? Is that data encrypted, and if not, can it **be** encrypted?" I asked. Every time the answer was the equivalent of "I'll need to get back to you" – and almost universally the answer was that the data was not encrypted when stored on the backup company's servers. Or, when it was encrypted, the backup company's staff had access to the encryption keys. Which, for the record, is effectively the same as storing the data unencrypted… at least in **my** book.

(CONTINUED ON PAGE 5)

Peter Pontificates... (Cont.)

(CONTINUED FROM PAGE 4)

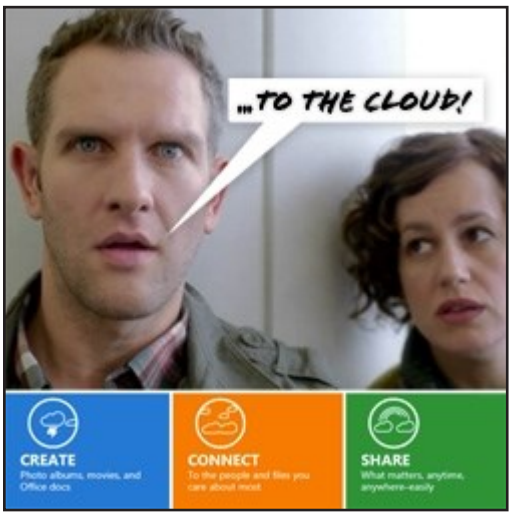
How can **any** company larger than a roadside lemonade stand be willing to store their private business data on servers owned, operated, and entirely controlled by a group of people they know nothing about? To me, it just defies common sense. So, I asked the IT consultants we hired.

"They're SAS 70 Certified," they replied. "That means they're entirely secure and trustworthy!" So I spent some time investigating this mystic SAS 70. And guess what? It ensures nothing, absolutely and totally nothing, about the security of the data you store in a SAS 70 Certified data center.

We've got tons of source code. This includes source code for our own products, projects we've done for clients, and even code given to us as examples. Some of this code we own. Some of this code is owned by others. We've got accounting records, detailing who our customers and suppliers are and what we've charged and paid them respectively. If I backup this data onto servers in our office, I can lock the server room. I can take copies of backups home and put those copies in my safe (and yes, I *do* have a safe at home. Actually, I have more than one). I can send copies of those backups to OSR's office in Vancouver, where they can be similarly safely stored under lock and key and protected by staff members who we, as a company, trust because they have been properly and thoroughly vetted.

But how could I trust this data to some data center, about which I know absolutely nothing? Who says they know what they're doing? Who says my data will be secure? Heck, who'll even assure me that the next Snowden with a grudge against one of our clients doesn't happen to work at the data center holding my "cloud-based" backups? Of course, there are no such assurances. The best assurance you can get is "trust us, because we care about your data." OK, but as they say, Доверяй, но проверяй.

Yet thousands, maybe zillions, of companies do just this. Trust. And they don't verify. Because they really can't. Or they do worse. They do much, much worse, in fact. Not only do they blindly trust a group of unknown ass clowns with their backups of their vital business data, they outsource **their email**.



If you're company is like ours, we use email to discuss everything from the trivial to the critical. Would any of you want the contents of you corporate email posted on the Internet? No? Couldn't you be doing exactly that by using an email system hosted "in the cloud" by a third party? I would say so. Yet more and more companies are moving their email services "to the cloud."

Let me give you an example. There's a company with whom we have a long-standing technology partnership. This multi-billion dollar high tech company, which shall remain nameless, works in a highly competitive field. In their field "the next big thing" is often a bet involving, very literally, billions of US dollars. The people working there are smart, technically sophisticated, and insightful. They have shown the ability to innovate and predict evolving industry trends. Heck, they even sometimes **create** evolving industry trends.

They also outsource all their email.

Seriously. A few years back, they moved the entire company's email system to an external, cloud-based, hosted solution. So, as we're exchanging product plans, schedules, marketing documents, presentations, and source code with them, that data is all being stored in "the cloud." By a third-party. Where it's probably guarded by a cadre of over-worked and underpaid, part-time, hung-over, college students listening to Avicii and wondering if they'll be able to score some Jägermeister and extra football tickets for the coming weekend.

I asked one of their senior folks about this once. His answer? "Yeah, I don't know. IT says it's OK, and that's what they've decided to do. It saves money or something, I guess. So…"

Did I say **I just don't get it** yet? Doesn't this seem like an awfully big risk to anyone besides me? The willingness of sane folks to commit their confidential and proprietary data to a random data center in a random location with random safeguards seems to me to be one of the craziest ideas I've heard of since train surfing.

(CONTINUED ON PAGE 31)

SYN, SYN, ENQ
Understanding Sync Scope in WDF Drivers

Serialization, the ability to do one or more of lines of code atomically, is a vital issue in WDF drivers. In fact, due to the inherently pre-emptible, reentrant, and multiprocessing nature of Windows, getting synchronization right is critical for any kernel-mode module.

WDF provides numerous options for serializing access to shared data structures. One of those options is Synchronization Scope, which most people refer to as "Sync Scope." In this article, we're going to examine the concept of Sync Scope. We'll talk about how it works, the complexities it introduces, and when it might be the most appropriate option for implementing serialization in a WDF driver.

What's The Problem?

Just to be sure we all understand the problem that proper serialization solves, let's look at an example. If you're already well versed in writing multi-threaded/multi-processor safe code, feel free to skip this section. If you wonder about when and why we need proper serialization in Windows drivers, read on.

```
typedef struct _MY_DEVICE_CONTEXT {

    WDFDEVICE   DeviceHandle;
...
    //
    // Statistics
    // We keep the following counts/values for our device
    //
    LONG        TotalRequestsProcessed;
    LONG        ReadsProcessed;
    LONGLONG    BytesRead;
    LONG        WritesProcessed;
    LONGLONG    BytesWritten;
    LONG        IoctlsProcessed;
...

} MY_DEVICE_CONTEXT, *PMY_DEVICE_CONTEXT;
```

Figure 1 – Example statistics in Device Context Block

Suppose we maintain a block of statistics in our driver's Device Context such as shown in **Figure 1**.

To maintain these statistics, our driver would increment and update the appropriate files in each of its appropriate EvtIoXxx Event Processing Callbacks, as shown in **Figure 2** (Page 7).

On entering this routine, we get a pointer to our Device's context structure, and then get a pointer to the user's data buffer and its length. We then update all three relevant statistics fields. After this, we presumably go on to do the rest of the work this driver needs to do.

But there's a problem with the statistics code in this routine as written. That problem is that there's no serialization performed when the statistics are updated. Consider what could happen if we get unlucky, and our EvtIoRead function (for the same device) is running at the same time on two processors and tries to update the statistics block. It's possible that we get the following sequence of events:

1. The code running on Processor 0 reads the value for **TotalRequestsProcessed** from the Device Context. Let's say that value is currently 0.
2. Right at the same time, the code running on Processor 1, which also is executing our EvtIoRead function and trying to update the statistics, also reads the value for **TotalRequestsProcessed** from the Device Context. It's also going to read a value of 0.
3. The code on Processor 0 increments the count, and writes it back to the Device Context. **TotalRequestsProcessed** is now 1.
4. The code running on Processor 1 does the same thing: It increments the count it read, and writes it back to the Device Context. **TotalRequestsProcessed** is now (still) 1.

The result from the above sequence of events is obviously incorrect. The reason it's wrong is because both copies of the EvtIoXxx read the original value for **TotalRequestsProcessed** at the same time (or, very close to the same time), add 1 to it, and write it back.

The way we fix this problem is to apply serialization. We guard the device statistics in our Device Context with a lock. This lock has the characteristic that it can only be held by one thread at a time, and any other requestors who try to acquire the lock while it's held by another thread wait until the lock is available. We make a rule in our driver that anytime you need to update the data in

(CONTINUED ON PAGE 7)

Understanding Sync Scope... (Cont.)

(CONTINUED FROM PAGE 6)

the statistics block, you must be holding the lock. Thus, the new steps would be as follows:

1. The code running on Processor 0 successfully acquires the lock guarding the statistics block.
2. The code running on Processor 0 reads the value for **TotalRequestsProcessed** from the Device Context. Let's say that value is currently 0.
3. The code running on Processor 1 (which also is executing our EvtIoRead function and trying to update the statistics) attempts to acquire the lock guarding the statistics block, but waits because the lock is already held by the code running on Processor 0.
4. The code on Processor 0 increments the count, and writes it back to the Device Context. **TotalRequestsProcessed** is now 1.
5. The code running on Processor 0 releases the lock
6. The code running on Processor 1, which was waiting for the lock, now acquires it (now that the lock has been released by the code running on Processor 0).
7. The code running on Processor 1 reads the value for **TotalRequestsProcessed** from the Device Context. It read the updated value of 1.
8. The code running on Processor 1 increments the count it read, and writes it back to the Device Context. **TotalRequestsProcessed** is now 2.
9. The code running on Processor 1 releases the lock.

And the result is now… correct. Yay!

```
VOID
MyEvtIoWrite(WDFQUEUE    Queue,
             WDFREQUEST  Request,
             size_t      Length)
{
    NTSTATUS                status = STATUS_SUCCESS;
    WDFDEVICE               device = WdfIoQueueGetDevice(Queue);
    PMY_DEVICE_CONTEXT      context;
    PVOID                   buffer;
    PVOID                   length;

    //
    // Get a pointer to my device context
    //
    context = MyGetContextFromDevice(device);

    //
    // Get the pointer and length to the requestor's data buffer
    //
    status = WdfRequestRetrieveInputBuffer(device,
                                           MY_MAX_LENGTH,
                                           &buffer,
                                           &length);


    if (!NT_SUCCESS(status)) {

        // ... do something ...
    }

    //
    // Update our device statistics
    //
    context->TotalRequestsProcessed++;
    context->WritesProcessed++;
    context->BytesWritten += length;

    //
    // ... rest of function ...


    return;
}
```

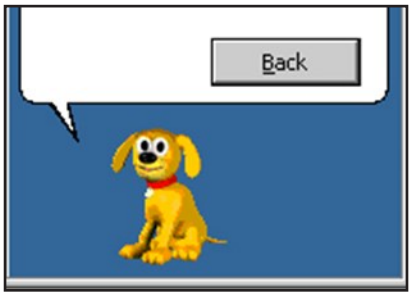
Figure 2—Keeping statistics in Device Context Block

(CONTINUED ON PAGE 22)

Map My Driver
Updating Drivers with KDFiles

Windows 2000 introduced Driver Verifier, which was the single greatest gift ever to be bestowed upon driver writers. This set the bar pretty high for what Windows XP would bring, but I must say it delivered by introducing three awesome features:

1. The triumphant return of Microsoft Bob's Rover the dog
2. A complete driver build environment. Remember what a mess it was when all you got were the headers and the libs from the WDK and the compiler and linker came from Visual Studio? Oh, wait…
3. Automatic driver replacement with KDFiles



It's Rover!

The first two features are now drifting off into history, but driver replacement using KDFiles is still supported and is, in fact, even more useful today (we'll get to why later). Unfortunately, we find that people either don't know about KDFiles or tried it but gave up after battling with the syntax.

In this article, we'll revisit the need for KDFiles and provide a bulletproof method for determining the correct mapping syntax for driver replacement.

Why Driver Mapping?

Developing drivers on Windows requires two machines: a host machine for development and a target machine for running the driver under test. We never, **ever** want to run our driver on our development machine for fear of corrupting our system from the incessant crashing of our buggy driver.

This then leads to the practical problem of copying a driver that we built on our development machine to our target machine for testing. Without KDFiles, there is no maximally convenient way to do this. You can copy the file over the network or via a USB drive, each of which has their own annoyances. Virtual Machine debugging has the benefit of drag and drop, but that assumes that VMware hasn't suddenly decided that it doesn't want to do that anymore. And, of course, all of these assume that you actually **remember** to copy the file over. Nothing kills the satisfaction of fixing a bug more than rebuilding your driver just to restart your tests using the old version.

KDFiles provides the ability to automatically update a driver on the target with a new file from the host each time the driver loads. This means you can always be running the latest version of your driver on the target without any additional post build steps. All you need to do is use the **.kdfiles** command to provide WinDbg with a "replacement map", indicating which drivers on the target should be replaced by files from the host. The target O/S and WinDbg will then do the work to copy the updated driver over the kernel debug connection on each driver load.

Registering Replacements with Mapping Files

There are a couple of different ways that you can indicate a replacement mapping, but by far the most convenient is through a Mapping File. This is a simple text file with the following syntax:

```
map
Module path of driver on target
File system path of driver on host
```

The Mapping File can contain multiple mappings, each of which is preceded by the mandatory **map** keyword.

The second component of the mapping file is the tricky one. This is where we must specify the module path of the driver on the target, which is **not** the same as the driver image's location on disk. Some examples of common, equally valid module paths are:

- \Windows\System32\drivername.sys
- \SystemRoot\System32\drivername.sys
- \??\C:\Windows\System32\drivername.sys

(CONTINUED ON PAGE 9)

Updating Drivers with KDFiles (Cont.)

(CONTINUED FROM PAGE 8)

The path used greatly depends on how the driver was installed and the current data of its **ImagePath** value. This effectively makes it impossible to know **a priori** what the module path of the driver will be on the target.

This is by far the number one usage problem with KDFiles and we've never found a documented, foolproof way of determining the module path. There is however an undocumented way that is guaranteed to work until, well, it doesn't, which is what we'll demonstrate here.

KdPullRemoteFile

The undocumented API **KdPullRemoteFile** is called by the target O/S during each driver load. This API is responsible for querying the host to determine if a mapping exists and, if it does, transferring the data to the local machine. By way of observation, this API takes as the first parameter a pointer to **UNICODE_STRING** that indicates the module path on the target. This is the string that must **exactly** match the path in your Mapping File, thus we can leverage this to determine what path we must specify in our mapping.

Let's see an example. Say we would like to replace the driver nothing_kmdf.sys with a file from the host machine's local disk. The key piece of information that we need is the module path of the driver on the target. To find this information, we can first set a breakpoint on **KdPullRemoteFile**:

```
kd> bp nt!KdPullRemoteFile
kd> g
```

We then need to trigger a load of the driver, for example by disabling and then enabling the device. This hits our breakpoint, at which point we can decode the first parameter using the Display Unicode String command (**dS**).

```
Breakpoint 0 hit
nt!KdPullRemoteFile:
fffff802`35870d9c mov     rax,rsp
kd> dS @rcx
ffffcf81`50282fa0  "\SystemRoot\System32\drivers\Not"
ffffcf81`50282fe0  "hing_KMDF.sys"
```

Note that for an x86 system, the first parameter would be located on the stack at ESP+4 and not in RCX.

We can now generate the Mapping File by saving the following to a text file:

```
map
\SystemRoot\System32\drivers\Nothing_KMDF.sys
e:\nothing_kmdf.sys
```

(CONTINUED ON PAGE 21)

This Time for Real
Windows and Real-Time

By Daniel Terhell
Community Contributor

Windows is not a real-time operating system is a phrase that's often echoed on the NTDEV forum. Frequently it comes up when someone runs into trouble trying to write a Windows driver for a device that's not designed with Windows compatibility in mind, such as a device that expects the software to respond within a short timeframe.

The defining characteristic of a real-time operating system is that it offers predictable execution that can meet deadline requirements. It must offer absolute determinism by giving guarantees of being capable of responding to requests within short timeframes. Often, a distinction is made between *soft* real-time and *hard* real-time environments. For soft real-time, responsiveness is highly desired but a missed deadline does not count as total failure. In hard real-time environments, by comparison, there is no tolerance for unexpected latencies and missing any deadline means total failure. Often such environments are responsible for handling life critical tasks.

On Windows, all requests to the operating system are processed on a best-effort basis, without any response time guarantee. While Windows is certainly capable of servicing hundreds of thousands of large requests per second, in general, it won't be able to guarantee that every single request is processed within a specifically defined short timeframe.

Windows is designed and optimized with performance and throughput in mind for general purpose uses, not for real-time tasks and low latency, which often have conflicting interests. Whereas a solution that is optimized for general purposes cares about things such as average throughput performance, a solution that has real-time requirements instead cares about maximum response times, the worst cases that can possibly lead to a deadline being missed. This means that every individual request or operation matters.

There exist many types of solutions that may have real-time requirements of some sort, ranging from industrial, medical, military, aviation, research, and high precision to multimedia applications. A common example of an application that has real-time demands is "real-time audio", such as a software synthesizer that needs to produce sound in response to keys being pressed on a MIDI keyboard with no more than a few milliseconds latency. An audio stream must be continuous and any interruption of the audio stream has audible consequences recognized as clicks, pops and drop outs. This means all requests must meet their deadlines.
It's interesting to note that the problems of latency in real-time applications as well as the "key press before sound" latency problems are ancient and not merely artifacts of the computer era. Organ players, for example, must read ahead and hear back what they play with many seconds delay after the air has been processed through the pipes using sophisticated mechanisms and the sound has travelled from the corner of the church to the ear.

Latency problems in real-time audio applications occur often as software synthesizers and audio plugins most often have their logic implemented in user-mode. This means they are more exposed to possibilities of interruptions (e.g. paging, scheduling) than a device driver that runs at elevated IRQL.

Finally, note that for the purposes of this article, we'll ignore certain Windows features such as IRQL PASSIVE_LEVEL interrupt handling and UMDF which, while handy for certain specific uses, are generally unsuitable designs for real-time processing.

Enemies of Low Latency

In order to better understand why Windows is not always a suitable operating system for real-time processing, we will first take a closer look at the path of an interrupt-driven device with deadline requirements that has its requests processed in user-mode. We will then examine possible problems that can cause undesired latencies on the way.

For a real-time sensitive device that is interrupt driven, it is essential that Windows can respond to requests in a timely manner. Because of the enemies of real-time processing discussed later, latency can be introduced during the processing path. Logically, during the processing of an interrupt, the lower the IRQL drops as processing proceeds from ISR to DPC and then on to user process servicing, the higher and more frequent latency spikes are to be expected.

(CONTINUED ON PAGE 11)

Windows and Real-Time (Cont.)

(CONTINUED FROM PAGE 10)

Interrupts/ISRs

A common way for a device to indicate that it requires attention from the software is to generate an interrupt. After a device fires an interrupt signal, it is received by a CPU which will execute the Interrupt Service Routine (ISR) that was registered for that interrupt.

The execution of the ISR can be delayed by various factors. The CPU may be temporarily deferring interrupts, the operating system could be servicing a higher level interrupt, temporarily have its maskable interrupts disabled as a result of some system wide lock state or otherwise. Also, the interrupt may be delayed by hardware factors. Unfortunately it is not possible to measure such interrupt latencies through software alone. It would require assistance of the hardware or by using a bus analyzer [*In modern Intel architecture systems, we know that this hardware latency is very low, almost always less than one microsecond – Editors*]. In this article, we will only discuss latencies during the software part of interrupt processing, that is, from the moment an ISR started executing until the moment the interrupt got serviced by the software.

As Windows does not allow prioritization control of device interrupts, at any time while an ISR routine is executing it can be preempted by a higher level interrupt unless it takes measures to prevent such preemption by raising IRQL. Latencies during ISR processing can also be introduced by other factors such as the operating system interrupt processing of the system clock, Inter Processor Interrupt (IPI) routines or even factors beyond the control of the operating system such as System Management Interrupt (SMI) routines and various hardware factors which will be discussed later.

DPCs

If the servicing of the interrupt demands lengthy processing, then the ISR typically schedules a Deferred Procedure Call (DPC). This DPC continues processing the I/O operation at a lower IRQL where device and system interrupts can preempt its execution. A DPC is also required if interrupt servicing needs to take place in user-mode, as it's not allowable to resume waitable objects or wake up user-mode threads under the IRQL restrictions imposed on ISR routines.

Generally, the DPC starts executing immediately on the same processor as it was requested on. The operating system maintains a DPC queue per logical processor and it may be possible that other DPC routines will execute first before yours when the DPC queue on your processor is being drained, thus adding latency.

Because DPC routines execute at IRQL DISPATCH_LEVEL (an exception to this is threaded DPCs which execute at PASSIVE_LEVEL), latency can also be introduced by any type of device or hardware interrupt that can occur during the processing of the DPC.

(CONTINUED ON PAGE 16)

Isolation Realized
Project Monadnock

Over the past several years, we have promoted the isolation filter model as a solution to a number of data virtualization projects. While we haven't returned to discussing isolation recently, we have gained considerable experience building isolation filters. One of these projects that we are excited to be telling you about is our Monadnock Project, named after the most popular mountain to climb in the US.

For the past ten years, we have been working on the problems of transparent encryption and compression in a Windows environment. It's been an amazing experience and our understanding of the complexities and nuances of the environment have continued to grow. In addition, the world has changed dramatically in the ensuing years. Data encryption isn't something just for Windows any more and solutions need to be able to work across numerous platforms.

Even on Windows we know there are challenges. For example, interactions with the native file systems can lead to unexpected performance behaviors. And those folks in Redmond sure seem to work hard thinking up new ways to make our lives more difficult as each new Windows release contains new and interesting features to which we must adapt.

So a couple years ago we started planning a new version of our data transformation toolkit and after quite a bit of time and planning – along with diversions to deal with the complexities of the WHCK tests and do other activities necessary to make a living – we reached the conclusion that what we needed wasn't a new version of our existing toolkit, but rather a *different* product. The key features we saw that our customers needed were:

- **Encryption focus**. As much as we love compression, there was no way to avoid the harsh reality that it is expensive (performance wise) to support and complex to implement. Plus, it turns out none of our customers are using it. Encryption is where it's at.
- **High performance**. In the past we had added some cool features, like alternate data streams for all file systems, but that in turn caused a lot of extra I/O overhead and a corresponding drop in performance. Given that customers didn't really care about streams for their products it doesn't make sense to pay the performance penalty – not to mention the code complexity. So we decided it was time to simplify: focus on the features our customers need for building a robust, high performance encryption product.
- **Simpler programming model**. As much as we love kernel programming, it turns out that it's difficult to do correctly and our customers struggle to find and keep qualified kernel developers. So our goal was to build a framework in which it was possible to build an entire encryption product **without any kernel programming.**
- **Multiple platform support**. This is a vital part of building products these days: customers want to share data between their devices, be it PC, Mac, iPhone, Android device or even Windows tablet or phone. Of course, since some of these platforms lock us kernel developers out, we have to find new ways to provide at *some* measure of functionality.

Bottom line for us was the realization that this really was a different toolkit. It's very different than what we've done in the past. And yet it also allows us to leverage the insight and understanding we have gained in helping dozens of different customers implement their encryption products over the years. Thus was born **Project Monadnock**.

To better help you visualize the technology behind Monadnock, let's describe some of its key components in conjunction with the architectural diagram (see p. 13).

Monadnock: Isolation at Heart

At the heart of Monadnock is our isolation filter (which really is a mini-filter, albeit a complex one). While we do expect to make an isolation filter kit available in the near-term future, for now, we've put it through its paces in the Monadnock project.

(CONTINUED ON PAGE 13)

Project Monadnock (Cont.)

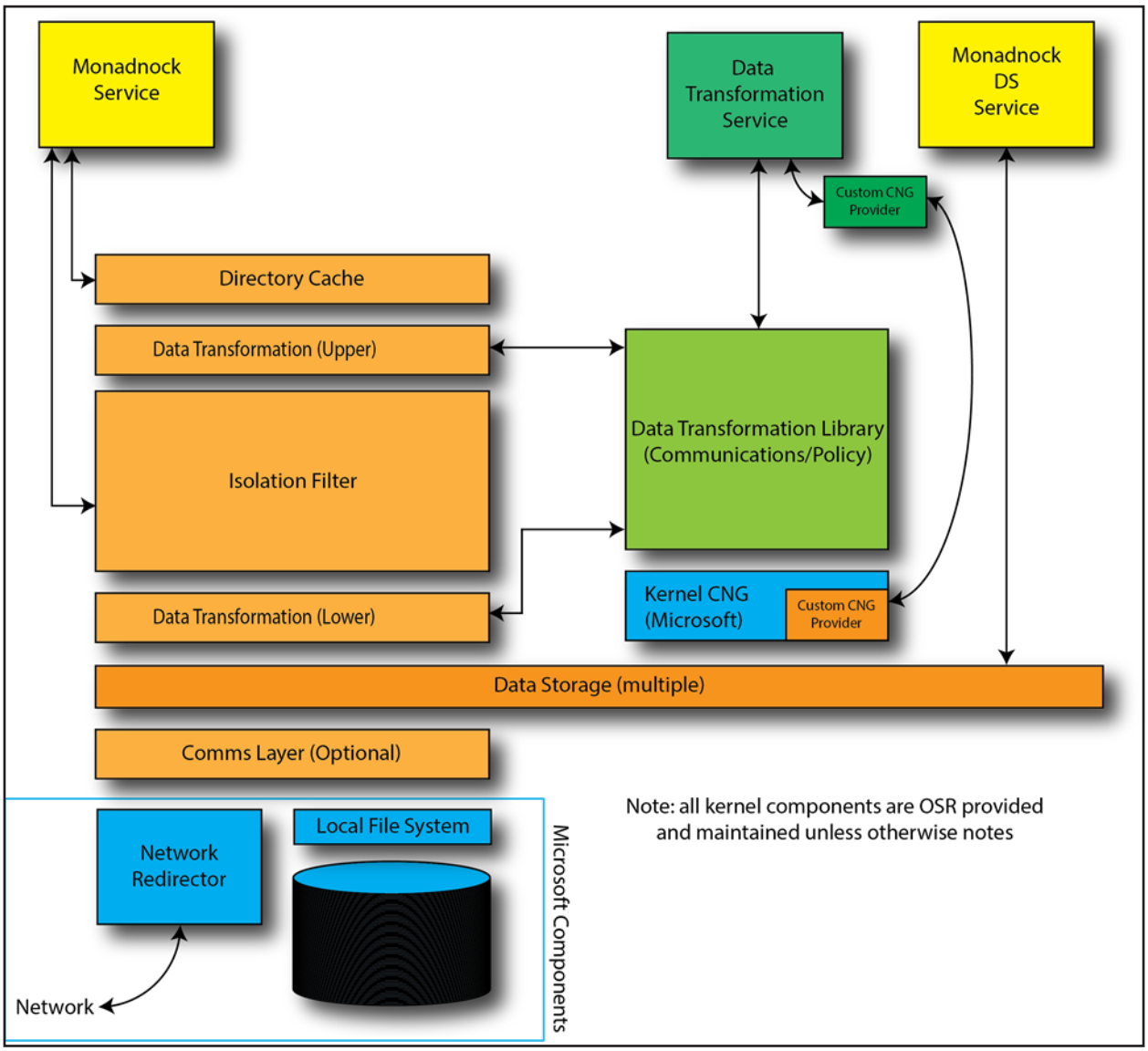
(CONTINUED FROM PAGE 12)

An *isolation filter* is a filter that focuses on isolating the "view" of a file from the "contents" of a file. From an implementation perspective, this is similar to the technique used by NTFS to provide transactional isolation: each such "view" is managed as a distinct virtual memory section – in this case, by the filter itself. It does this by managing the data in virtual memory and the file system data cache, so it's certainly *not* a simple mini-filter solution.

When an application opens a file, the responsibility of the isolation filter is to properly manage what that application sees as the contents of the given file. The specific choice of views is actually not something that the isolation filter does – this is deferred to other components. In the case of Monadnock, that would be the "Data Transformation" layer. While isolation permits multiple views per file, Monadnock uses just two views: the "encrypted" view and the "decrypted" view. Thus, Monadnock based solutions can choose, on a per open basis, what exactly an application will "see".

For example, suppose you are building an encryption product that wants to send documents attached to an Outlook e-mail in their encrypted form but you also want to ensure the Outlook files are themselves encrypted. In that case, the policy would be to give "encrypted" views to files other than those identified as Outlook files – such as the "OST" and "PST" files that Outlook uses for its cache and archives. Such decisions can be made based upon various attributes of the file open operation including the fully qualified path name of the file, the access requested, the disposition requested, the process and/or thread opening the file, the security credentials of the thread and/or process. This gives considerable control to those building real-world products with Monadnock.

(CONTINUED ON PAGE 14)



Monadnock Architecture Diagram

Project Monadnock (Cont.)

(CONTINUED FROM PAGE 13)

Directory Caching

One of the challenges in building any filter on Windows is that applications frequently rely upon the *size* information within a directory listing in order to determine "how big" a file is. But because encryption changes the size of a file by changing its padding and adding additional "header information", failing to provide the correct size information to applications can lead to incorrect behavior.

While it is possible to "correct" these entries on the fly, our experience in the past tells us that it's important for performance reasons to cache these size corrections in order to minimize their impact on application performance. Every unnecessary I/O that we introduce in the system leads to dramatically lower performance. Thus, our directory caching layer is optimized to ensure we do not open files that aren't encrypted *and* to cache both the physical and logical size information for files that are encrypted.

Key Management and Encryption using CNG

Another important decision for us was to use the Cryptography API: Next Generation. This library is supported with Vista and more recent and permits use of existing encryption implementations (notably AES) as well as the many optimizations that Microsoft has included (AES-NI instruction support for those CPUs that can exploit it, for example). CNG has also been certified to US FIPS 140-2 standards, making it easier for our customers to qualify their products for Common Criteria certification, for example.

CNG can be used from both kernel mode as well as user mode and Microsoft documents how to build extensions to CNG, components known as Cryptographic Providers. These providers can implement not only encryption algorithms but also Key Storage Providers.

Our goal in choosing this paradigm was to further simplify development for those using the Monadnock framework, while also making it easier to build robust, performant products that meet stringent security requirements for current and future environments.

Flexible Data Storage Layer

In our previous work we had a *very* feature rich data storage layer. After analyzing how our customers were using it, we determined that most of them did not need many of the features that we were providing, but they were paying a performance penalty associated with those features. So, while we love technically rich projects, we decided to look at ways to simplify our data storage model, while at the same time permitting those customers that do need to support additional features the flexibility to do so.

Thus, Monadnock's data storage layer is a simplified, minimal state interface where, by default, we defer complex processing to a user mode service. For example, when an existing file should be encrypted, rather than perform that complex work in kernel

(CONTINUED ON PAGE 15)

Project Monadnock (Cont.)

(CONTINUED FROM PAGE 14)
mode (as we did in the past) we now refer the operation to a user mode service.

The user mode service can then take advantage of things like the ability to display "progress bars" to the user encrypting a file and opening the file for exclusive access while the conversion is ongoing. This has the added benefit of keeping the kernel mode implementation simpler, which in turn translates into a more resilient end user product.

One key to our model, is to use a shared source library for data storage management between user mode and kernel mode. Thus, a subset of the same library is used in kernel mode to access the physical contents of the file, ensuring compatibility between the user mode and kernel mode code. This technique also greatly simplifies the process of developing this layer, as most of the development and debugging work can be done in user mode.

In addition, this has also permitted us to structure this critical library in such a way that we can support non-Windows platforms. Thus, this is a key piece permitting interoperability between platforms.

Shifting from Kernel to User

One thing we have seen in working with prospective customers is that frequently they do not have kernel level expertise. Our existing customers often tell us they have a difficult time finding such individuals and when an organization does invest to train someone, they often leave the organization because they can do better elsewhere.

We live, eat and breathe kernel development at OSR, so it took a while for us to wrap our heads around the idea of pushing more processing work into user mode. So while we might be slow, we have realized that in order to make Monadnock easier to use we have to make it possible to use it **without any kernel programming at all**. Thus, a developer using the Monadnock framework can leave the kernel programming to us, while they focus on adding the key management, administration, and other unique product features, all in user mode.

Of course, for customers that need to have control of the kernel components, they can certainly do so, as Monadnock will remain available (under license) in full source code form.

Developer Version

In the past, we've always struggled with the idea of providing developer versions of toolkits. This was at least in part due to the difficulty in defining "best practices" example implementations that matched the widely varying policies and product features that customers were asking to see. The shift in focus to enabling customers to develop in user mode gives us a much clearer model for providing a base example, and allowing developers to "try out" their own customizations to Monadnock to match their product needs.

Thus, we will make a developer's version available for Monadnock. We will include source code to the user mode components we expect developers will customize, with the other components provided in binary form, either as drivers, executables and binary libraries.

Monadnock: Our Newest Framework

We're excited about Monadnock, as it will broaden the appeal of our encryption solutions far beyond what we've been able to accomplish in the past. For our existing customers, it offers a path forward as we continue to evolve and improve our product offerings.

We expect to make Monadnock available in Beta form by the end of September 2014, with a final release by the end of December 2014. The full release will still focus on Windows, but it will include non-Windows platform support, which we will continue to enhance as we learn more about our customer's needs.

For those interested in joining the Monadnock Beta program, please sales@osr.com.

Follow us!

Windows and Real-Time (Cont.)

(CONTINUED FROM PAGE 11)

The time interval from the moment an interrupt is received on a CPU until a DPC routine starts executing is often referred to as "ISR to DPC latency". If the driver has deadline requirements it will want to minimize the worst case ISR to DPC latencies. This is an important point, don't miss it: For real-time processing what we care about minimizing is the **worst case latency**, and not the **average** latency. Average ISR to DPC latency is almost always very good. On the other hand, worst case ISR to DPC latency can sometimes be astonishingly bad. And this is where artifacts are most often introduced (See sidebar, *Play By the Rules*, below)

User-Mode

Executing user-mode code in the request processing path of a real-time solution should generally be avoided. However there are situations and solutions that require this. As the user-mode code executes in a critical path, obviously that code should execute as quickly as possible. The user-mode code should preferably not block, wait or even call any Windows API library functions at all. It should only make use of resident memory as hard pagefaults can cause long suspensions of execution of a thread while the page fault handler needs to resolve it by synchronously reading in data from disk, which can take seconds.

Note that Windows API functions such as **VirtualLock** only allow an allocation to be assigned to the working set of a process and do not provide the application with memory which is guaranteed to be resident in RAM. A sure way to provide an application with resident memory is to have a driver lock buffers provided by the user into memory. There are numerous ways to do this, some more complex than others. One simple method is for the user application to allocate the buffer and provide it to the driver as the OutBuffer in an IOCTL using METHOD_OUT_DIRECT. The driver then keeps this buffer – with the locked memory – in progress until the application exits. There are other, more complex, ways of locking shared memory between user-mode and kernel-mode as well. The advantages and disadvantages of each are outside the scope of this article.

User-mode threads that execute as part of interrupt processing are usually kept in a pool of real-time priority threads suspended in an idle wait state and woken up by a dispatcher object signal such as an event or I/O completion, set from a DPC routine.
The time interval from the moment an interrupt was received on a processor until a user-mode thread starts executing after having been woken up is often referred to as "ISR to process latency". This includes the time required to schedule and execute a DPC routine as part of this process. In operating system courses this is sometimes referred to as "Process Dispatch Latency". A solution with deadline requirements that does part of its processing in user-mode must minimize its ISR to process latencies.

In case a real-time critical user-mode thread needs to execute for extended periods of time, the scheduler of the operating system makes tricks available that allow non-preemptive scheduling of a user mode thread by giving it a "super" real-time priority so that its quantum never expires. This involves assigning the process to a job object with specific scheduler class settings.

(CONTINUED ON PAGE 17)

Play By the Rules

On a side note, as a kernel developer, even if your driver has no real-time requirements whatsoever, you should still care to play fair and avoid spending too much time at elevated IRQL so that you won't compromise the real-time capabilities of the system that your driver is running on. That includes time spent in ISRs, in DPCs as well as code executed while a spinlock is held or the IRQL is raised through other means. Or if you are a BIOS/firmware developer that counts even more.

The MSDN guidelines on this are that no DPC or ISR routine should ever be executing for longer than 100 µs. In practice, many drivers are violating this rule, at least when run on certain hardware. If a driver is causing high latencies, it doesn't necessarily mean it's the software at fault, in many cases such problems can be fully attributed to the hardware. Network drivers (WiFi in particular), storage port drivers and ACPI battery drivers are among the most notorious for causing high latencies during interrupt processing. Often such drivers need to be disabled when configuring a system to handle tasks with real-time demands.

Windows and Real-Time (Cont.)

(CONTINUED FROM PAGE 16)

More Zombies

Apart from problems with hardware interrupts, ISR routines, DPC routines, user-mode processing and hard page faults, there are some other enemies of low latency under the hood that cannot be neglected, as they can cause significant delays. Let's discuss some of these now.

Inter Processor Interrupts are OS initiated interrupts that block all device interrupts while a specified routine is executed on all logical processors. They can be triggered by both hardware and software. Drivers such as ndis.sys make use of this technique which, along with heavy duty DPC processing, is one of the reasons why systems configured with networking enabled are often not suitable for real-time processing.

System Management Mode (SMM) is a special-purpose operating mode of the processor provided for handling system-wide functions like power management, system hardware control, or proprietary OEM provided code. It is intended for use only by the BIOS or firmware interface, not by applications or the operating system. System Management Interrupts are interrupt routines that the CPU executes in SMM beyond the control of the operating system. SMIs take precedence over any other maskable or non-maskable interrupt. Since an SMI can interrupt any processing path at any time, it needs to execute as quickly as possible or it will render an entire system unusable for processing real-time tasks.

A CPU core with a variable speed setting activated can reach a high temperature trip point after which it is temporarily kept in a "stop-clock mode" for several milliseconds to have it cool down. Interrupts occurring on this processor will be deferred until the CPU is allowed to run again.

Unexplainable CPU stalls cay also be caused by bugs in the design of the CPU. Several CPU bugs in modern CPUs are related to processor C-states (which implement CPU idle behavior) and P-states (which implement CPU speed changes) and can freeze a processor indefinitely until certain conditions are met. CPU manufacturers publish errata (spec updates) with such information.
Many of the factors which cause latencies that have been mentioned are impossible to control from software by a developer without controlling the system configuration. It remains a possibility however to measure most of the capabilities of a system for real-time processing through software so that an end-user can be notified in case his system appears not to be compliant.

Measuring Latencies

We have outlined a great number of potential dangers for a driver with real-time requirements. If a driver is to support an off-the-shelf piece of hardware that has any sort of real-time requirements that is to run on an arbitrary system, basically all bets are off for its success. Therefore it may help to test the capabilities of a customer system before installation or acquisition to prevent failure and customer disappointment, or to help the end user overcome the obstacles of running your solution by helping him configure his system.

(CONTINUED ON PAGE 18)

Windows and Real-Time (Cont.)

(CONTINUED FROM PAGE 17)

There are several utilities that allow you to measure execution times of ISRs and DPCs. One of them is XPERF which comes with the optionally installed Windows Performance Toolkit. The Windows Performance Toolkit is included with both the Windows WDK and SDK. For more information about how to use XPERF, check out **The NT Insider** article *Get Low - Collecting Detailed Performance Data with Xperf* (see http://www.osronline.com/article.cfm?article=554).

Another utility is LatencyMon which reports ISR and DPC execution times as well as hard page faults. It also offers some different latency measuring tools including ISR to DPC and ISR to user process latencies. LatencyMon is written by the author of this article (see http://www.resplendence.com/latencymon).

As the author of the driver code, it is fairly trivial to add measuring points yourself by using the **KeQueryPerformanceCounter** function. To measure the execution time of your DPC routine, you can simply query the performance counter at the beginning and end of the routine and check the difference. You can use the same technique to measure ISR to DPC and ISR to user process latencies as well.

A technique that has been used in the past to measure "DPC latencies" was to install a periodic kernel timer of which the accuracy of the interval was measured. Timers in Windows are software based and are dependent on the resolution of the clock interrupt. The resolution of the system clock is a global resource that several software components and applications compete for in a way that only requests to lower the clock interval will be honored (i.e. the application with the lowest request wins). Because timers on Windows do not have direct support from hardware, it means that they are not very accurate. That is even more so on Windows 8 which introduces a new feature called "dynamic clock tick", where the system clock does not interrupt at fixed intervals but rather, when the operating system deems it necessary for power saving reasons. This has caused any measuring method dependent on a kernel timer to become unreliable. If a device should need accurate periodic attention from software, the hardware should come equipped with its own timer that can fire interrupts so that it can be serviced by software in a timely fashion. This is true despite the fact that the Windows 8.1 WDK has a new feature called high resolution timers.

Starting with Windows Vista, Windows offers a set of functions and classes to retrieve event tracing information which is collected by the operating system kernel. Among other things, this will allow you to obtain information about ISRs and DPCs as well as hard page faults executed in the system. Unfortunately, these classes do not allow you to collect information about time spent at elevated IRQL due to causes other than ISRs and DPCs, such as for example code running under a spinlock and IPIs.

One method used to measure the maximum execution times of SMIs and unexplainable stalls of the processor is to measure the highest interruption of a tight loop that spins at IRQL HIGH_LEVEL. This opportunistic method of polling will not, however, measure the execution of SMI routines initiated by software.

Similarly, you can measure the execution times of IPIs by performing the spinning loop at an IRQL of just below IPI_LEVEL. One thing to take into consideration here is that some versions of Windows can use an IRQL management technique known as “lazy IRQL.” In this technique the IRQL value stored by the system does not always correspond to the actual processor's task priority state.

Interrupts are bound to processors. While Windows allows full configuration over what processors in your system are to execute threads by setting affinities, there is little control over which processors handle device interrupts. The processor(s) that execute the ISR associated with a particular device is mostly dependent on the hardware, and therefore Windows normally honors the BIOS settings for interrupt affinity. Some chipsets cause interrupts to be spread across all processors while others cause interrupts to be exclusively executed on CPU 0. This makes it not very useful for software that is to run on arbitrary hardware to choose the processors on which it is to run through affinities. This situation is different if you have the option of supplying your own system which is discussed below. The **GetProcessorSystemCycleTime** API function is a quick and easy way to find out which processors in the system are actually servicing interrupts and DPCs.

Controlling the Hardware

As was mentioned previously, if you are developing drivers for a device or solution with real-time requirements, chances are that it may not meet its goals on certain systems it's deployed on, due to the specifics of those systems’ configuration.

(CONTINUED ON PAGE 19)

Windows and Real-Time (Cont.)

(CONTINUED ON PAGE 18)

If you have the luxury option of supplying a controlled configuration that includes all the hardware on which your solution is to be run then you have many options at your disposal to avoid failure. These options are not available to developers of an off-the-shelf product. Depending on the deadline and market requirements of your solution, choosing a specific system configuration may allow you to deliver an end solution that is guaranteed to work.

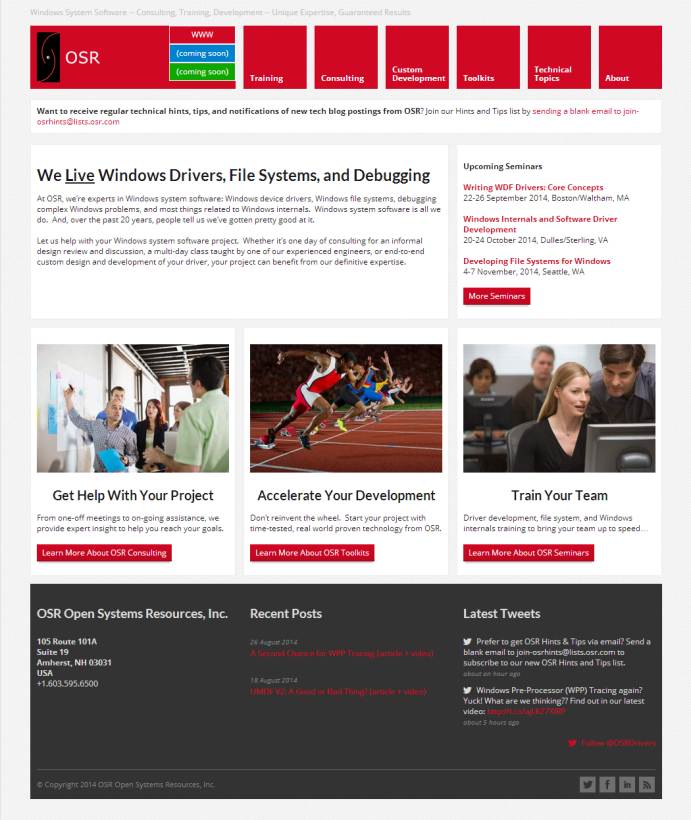
Once the system configuration is entirely under your control, you have the option of adding in additional hardware to assist your latency sensitive tasks. One option is to implement the real-time logic in hardware, for instance, by using a FPGA or DSP board. There are also real-time (software) extensions for Windows that allow you to meet real-time without additional hardware such as IntervalZero RTX and Tenasys Intime. These solutions work by running a separate operating system alongside Windows which will be responsible for executing the real-time logic.

But even without exotic hardware or real-time extensions there is a lot that can be done if you control the overall system configuration. If the required response time (or accuracy) of your solution is not very tight, say ten microseconds or longer, then you also have the option of configuring a Windows system with selected hardware and drivers that are known to not introduce high latencies in the system and be able to deliver a solution that is entirely running on Windows which is still guaranteed to meet its deadline requirements.

By carefully selecting the motherboard chipset and drivers in the configuration, it's possible to "control" what processors will be connected to interrupts, allowing you to reserve one or more processors for real-time critical tasks through affinities while effectively avoiding latencies induced by ISRs and DPCs. Control over what CPUs handle ISRs, DPCs and threads can be even taken further by booting the system with special parameters for processor group awareness testing.

An important part of a custom configuration is power management. As previously discussed, several power management features of the CPU and the BIOS can cause severe real-time processing latencies. Disabling these features, where possible, can avoid unsuitable latencies in real-time processing.

(CONTINUED ON PAGE 20)



OSR'S NEW CORPORATE WEBSITE

Not Because We Were Bored...

If only to keep ourselves insane, we have felt obliged to change up the look and feel of our website from time to time. But really, the new site is less about a new look, and much more about making it easy for us to maintain and add features to the site.

Of note, we're in the process of moving content off of the aging infrastructure of our resource portal (OSR Online). To date, this includes content from *The NT Insider*, and our "Dev Blog", but there will be more to come soon.

Windows and Real-Time (Cont.)

(CONTINUED FROM PAGE 19)

People in the audio industry have done lots of research on how to configure their Windows workstations for low latency so there is plenty of information available on the Internet. A good keyword here is DAW (Digital Audio Workstation). Also there are computer manufacturers who specialize in tailoring and delivering desktops and notebooks for low latency tasks running on Windows.

Summary

As you can see, getting a Windows system ready to be able to handle real-time tasks requires some measuring, puzzling, and even taking into consideration some factors that are quite uncertain. After having tweaked and configured a system along with your solution so that it works and having tested it, you may still feel uncomfortable providing a "guarantee" to your customer of which its proof requires analysis of statistics of measurements of operation.

Remember: Windows is not a RTOS. If your solution requires **guaranteed** response times from Windows, you **may** have your solution working most of the time, depending on the end user's configuration. The questions then become how frequently are your real-time requirements not met, what are the consequences of not meeting those requirements, and does any further tailoring or system configuration need to be done to improve the result. Hopefully this article explains the problems, how to overcome some of the issues, and what resources are available to you as a kernel developer who needs to deliver a real-time sensitive solution on Microsoft Windows.

Daniel started out programming at a young age on a ZX-Spectrum on which he soon learnt BASIC and assembly language and later C. After a career as an application developer he mainly specialized in system software and device drivers. Daniel is founder of Resplendence Software Projects which develops advanced system tools and developer components for Microsoft Windows. He can be contacted at danielrsp@resplendence.com

Updating Drivers with KDFiles (Cont.)

(CONTINUED FROM PAGE 9)

Next, we provide WinDbg with this replacement map by executing **.kdfiles** and specifying the path to the mapping file:

```
kd> .kdfiles e:\mapfile.ini
KD file assocations loaded from 'e:\mapfile.ini'
```

For sanity, we can execute **.kdfiles** again with no parameters to confirm the mapping:

```
kd> .kdfiles
KD file assocations loaded from 'e:\mapfile.ini'
\SystemRoot\System32\drivers\Nothing_KMDF.sys -> e:\nothing_kmdf.sys
```

The next time we load the driver, we should see output in the debugger indicating that the updated file is being "pulled" from the host to the target:

```
KD: Accessing 'e:\nothing_kmdf.sys' ...
KdPullRemoteFile: About to overwrite... Nothing_KMDF.sys and preallocate to 2670
KdPullRemoteFile: Return from ZwCreateFile with status 0
....
```

Voila! We're now running the version of the driver that was located in the specified directory on our host.

If at any point we want to clear the current set of mappings, we can use the **/c** switch to the **.kdfiles** command. Beware though that KDFiles **overwrites** the driver image on the target, so clearing the mapping does not actually revert the version of the driver.

The Boot Debugger Supports KDFiles Too!

We mentioned at the start that KDFiles is now even more useful, but how is that? Well, it turns out that the Boot Debugger also supports driver replacement. This means that you can automatically update your boot start drivers, which are otherwise a pain to update during development. To do this, you simply need to make sure that the boot debugger is enabled on your target machine using bcdedit:

```
bcdedit.exe /set bootdebug on
```

And provide a mapping for the boot start driver in your mapping file.

Conclusion

KDFiles is one of those things that we would now have a hard time living without, though it certainly isn't entirely user friendly in getting configured. If you've never heard of it before, you're certainly in for a treat. If you have tried but gave up, give it another shot with the steps we outlined in this article and let us know how it goes!

Understanding Sync Scope... (Cont.)

(CONTINUED FROM PAGE 7)

Why Proper Serialization is Hard

The concept that a given lock guards a series of fields in a structure is enforced only by the engineering discipline of the devs working on the driver. In other words, there's no Windows or WDF feature that can tell you "Hey, dude! You forgot to acquire the lock and you're updating this structure that's supposed to be protected!" While there **are** SAL annotations that can help with this, there is presently no 100% reliable method for enforcing your locking policy. Therefore, the onus is on you – the driver developer – to properly and consistently implement whatever locking policy you create.

Thus, there are several challenges to using serialization properly. Among them are:

- You, the developer, must recognize where serialization is needed in your driver code.
- Once you recognize where serialization is needed, you must choose the right type of lock (there are several different lock types) to use for that serialization.
- Having chosen the right type of lock, you need to be sure you acquire and release the lock properly, in every single place in your driver where serialization is required. Not locking when you should will lead to unpredictable results.

If you're not used to writing code that is multiprocessor and multithread safe, the above issues might seem daunting. And, trust me, getting your locking right **can** be quite a challenge, even for experienced devs. But, once you've gotten enough practice, identifying most potential "race conditions" (sequences that need to be performed atomically) will become second nature to you.

Introducing: Sync Scope

The above challenges are precisely the issues that WDF tries to solve with the concept of Sync Scope. Sync Scope provides a simple serialization scheme that seeks to relieve driver devs of some of the burdens of implementing proper serialization.

With Sync Scope, you simply tell WDF that you want the common callbacks associated with your **WDFQUEUE**, **WDFFILEOBJECT**, and **WDFREQUEST** to be serialized. As a result, the Framework will choose a lock type to use, and automatically acquire that lock before these routines are called in your driver and will release that lock when your driver returns from these routines.

You tell WDF the granularity of the locking that you want in your driver when you specify the Sync Scope. Your choices are simple:

- **WdfSynchronizationScopeDevice** – This tells WDF that you want all the applicable Event Processing Callbacks in your driver to be serialized at the Device level. This means that if you have multiple Queues per Device (say one Queue that handles both reads and writes and another Queue that handles device controls) you do not want any EvtIoXxx Event Processing Callbacks for the same device to be executing at the same time, regardless of the Queue with which the Callback is associated.

  When you choose Sync Scope Device, WDF maintains a lock at the Device level. Before calling any applicable Queue, Request, or File Object related Event Processing Callback, WDF acquires this lock. When your driver returns from one of these callbacks, WDF releases the lock. The locking structure is shown in **Figure 3**.

  So, in the scheme shown in Figure 3, none of the indicated EvtIoXxx Event Processing Callbacks (EvtIoRead, EvtIoWrite, EvtIoDeviceControl) for a given device would run in parallel. The Framework would automatically serialize

(CONTINUED ON PAGE 23)

Understanding Sync Scope... (Cont.)

(CONTINUED FROM PAGE 22)

these routines, acquiring a lock before they're called and releasing it afterwards. However, EvtIoXxx Event Processing Callbacks for WDFDEVICE 1 and WDFDEVICE 2 **could** run in parallel. The Framework is only serializing callbacks within a given device, not across all devices for a given driver.

- **WdfSynchronizationScopeQueue** – This tells WDF that you want the applicable Queue and Request Event Processing Callbacks to be serialized on a per Queue basis. So, if you have multiple Queues the Event Processing Callbacks associated with each specific Queue will be serialized. So, for example, let's once again assume you have one Queue that handles both reads and writes, and another Queue that handles IOCTLs. In this scheme, the callbacks for read and write will never run in parallel because they are serviced by the same Queue. However, either EvtIoRead **or** EvtIoWrite **can** be running at the same time as EvtIoDeviceControl, because read and write are handled by one Queue and device control requests are handled by a different Queue.

  When you select Sync Scope Queue, WDF maintains a lock for each Queue you create. Before calling any applicable Queue or Request related Event Processing Callbacks, WDF acquires the lock for the associated Queue. This ensures that any callbacks that are associated with a given Queue (and, hence, share the same lock) will never run at the same time. When your driver returns from one of these callbacks, WDF releases the lock. This is illustrated in **Figure 4**.

(CONTINUED ON PAGE 24)

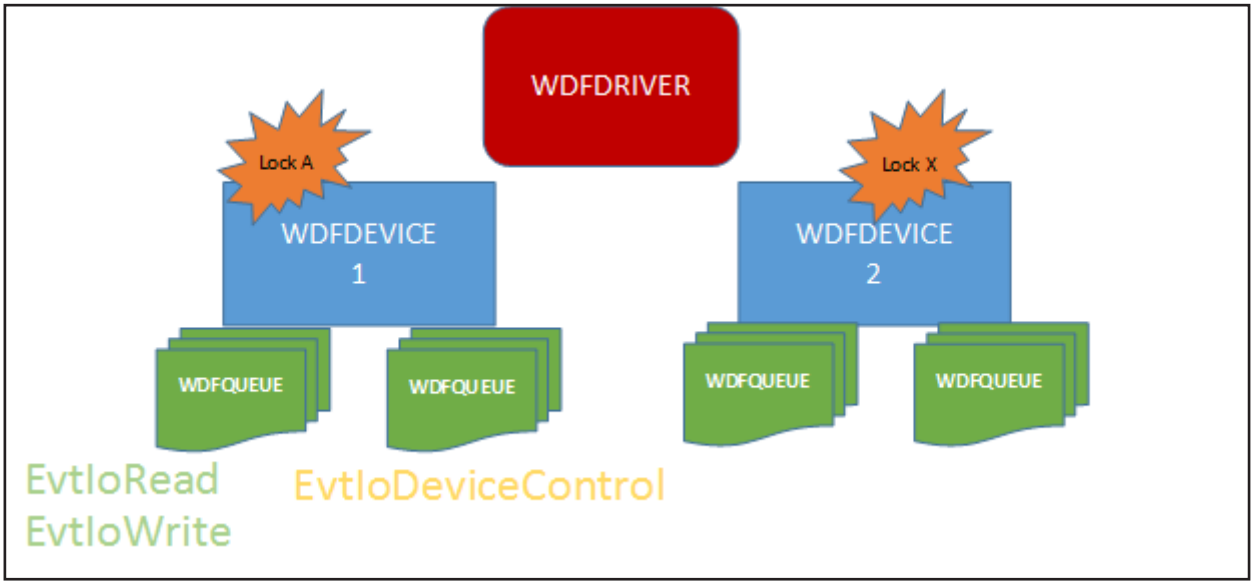


Figure 3 – Sync Scope Device: One WDF-managed lock shared by all Queues within a Device

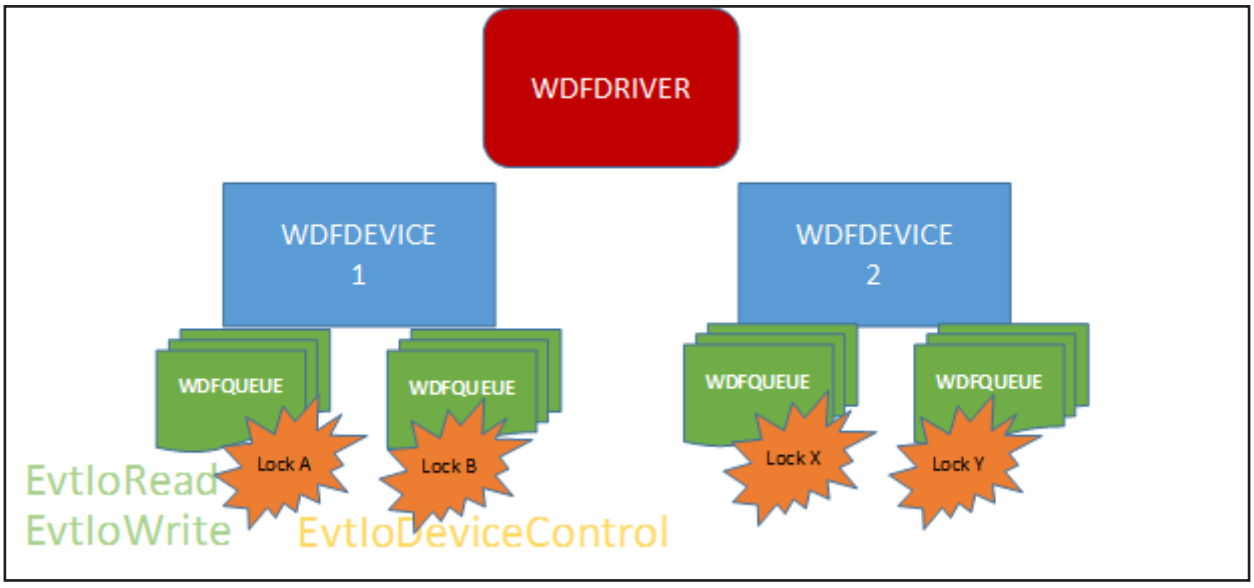


Figure 4 – Sync Scope Queue: One WDF-managed per Queue

Understanding Sync Scope... (Cont.)

(CONTINUED FROM PAGE 23)

```
// ... beginning part of EvtDriverDeviceAdd ...

WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE(&attributes,
                                        MY_DEVICE_CONTEXT);

//
// Ensure that all our EvtIoXxx callbacks never run in parallel
//
attributes.SynchronizationScope = WdfSynchronizationScopeDevice;

status = WdfDeviceCreate(&DeviceInit, &attributes, &device);

if (!NT_SUCCESS(status))
{

    // ... whatever ...
    goto Exit;
}

// ... rest of function ...
```

Figure 5 – Specifying Sync Scope Device

Thus, in the scheme shown in Figure 4, the EvtIoRead and EvtIoWrite Event Processing Callbacks would never run at the same time for a given device, because they are associated with the same Queue, and Sync Scope Queue has been specified. However, either of these callbacks (EvtIoRead or EvtIoWrite) could run at the same time as EvtIoDeviceControl. This is because WDF maintains the Sync Scope lock at the Queue level. And, of course, EvtIoRead (or any Event Processing Callbacks) for WDFDEVICE 1 could once again be running at the same time as EvtIoRead (or any Event Processing Callback) for WDFDEVICE 2. This is because locking is on a per Queue basis, and a given Queue is always associated with a single WDFDEVICE.

You specify the desired Sync Scope before creating either your **WDFDEVICE** or **WDFQUEUE** Object. Sync Scope is specified in the **SynchronizationScope** field of the **WDF_OBJECT_ATTRIBUTES** structure. For example, if you want the applicable Event Processing Callbacks in your driver to be serialized on a per-device basis, you specify **WdfSynchronizationScopeDevice** as shown the example code in **Figure 5**.

If you don't want to use Sync Scope Device, you can specify **WdfSynchronizationScopeQueue** in the **WDF_OBJECT_ATTRIBUTES** structure that is specified when you create any WDF Queue. This allows you to use Sync Scope Queue for only a subset of Queues if you so desire.

As to which Event Processing Callback functions are serialized, for Sync Scope Device the following functions are automatically handled:

- EvtDeviceFileCreate
- EvtFileCleanup
- EvtFileClose
- EvtIoRead
- EvtIoWrite
- EvtIoDeviceControl
- EvtIoDefault
- EvtIoInternalDeviceControl
- EvtIoStop
- EvtIoResume
- EvtIoQueueState
- EvtIoCanceledOnQueue
- EvtRequestCancel

(CONTINUED ON PAGE 25)

Understanding Sync Scope... (Cont.)

(CONTINUED FROM PAGE 24)

For Sync Scope Queue, all the above functions except those associated with WDFFILEOBJECTs (shown in blue, and named EvtIoFileXxx or EvtDevFileXxx) are serialized. Note that there are several interaction issues regarding Sync Scope and IRQL, which we'll discuss later in this article.

Love Sync Scope? Want More?

If you like the idea of Sync Scope, but you're thinking it would be nice if you could expand the functions to which Sync Scope applies, there's yet another feature you should be aware of: Automatic Serialization. You can optionally extend the serialization provided by either of the Sync Scopes to three other callbacks:

- The callback from a given WDFTIMER
- The callback for a given WDFWORKITEM
- The DpcForIsr associated with a given device

You can elect this option by setting the **AutomaticSerialization** field of the appropriate Object configuration structure to TRUE. To serialize execution of your DpcForIsr, you set the **AutomaticSerialization** field for the **WDF_INTERRUPT_CONFIG** structure to TRUE. In fact, it's worth noting that **AutomaticSerialization** is set to TRUE **by default** for **WDFTIMER** and **WDFWORKITEM** Objects. So your **EvtTimerFunc** Event Processing Callbacks and your **EvtWorkItemFunc** Event Processing Callbacks will, by default, be serialized at the Device level if you're using Sync Scope Device, and will be automatically be serialized at the Queue level if you make the Queue the parent of the created **WDFTIMER** or **WDFWORKITEM**.

Using Sync Scope: Life Is Good?

So, is Sync Scope the solution to all the serialization problems we could ever encounter in a driver? Maybe. It certainly simplifies your life as a driver dev. For example, regardless of the number of Queues configured, setting Sync Scope to Device prior to the creation of our Device Object will "fix" the statistics management example shown in Figure 1. If we parent any relevant Work Items or Timers on our Device Object, we'll even be able to safely manipulate the statistics in those callbacks, assuming we leave **AutomaticSerialization** set to TRUE when we create the Objects. And, if we needed to manipulate the statistics from within our DpcForIsr, all we'd have to do is set **AutomaticSerialization** to TRUE when we create our driver's Interrupt Object. So life is indeed easy. Using Sync Scope makes these serialization issues "just go away."

It's important to recognize that even if your driver has a single Queue and that Queue uses Sequential Dispatching, Sync Scope can still prove to be quite useful. For example, Sync Scope can certainly be helpful when you're using a single Queue with Sequential dispatching but have a dead man timer or background work item that could fire at any time while processing that one Request. Either of these activities (the timer or work item) could race with your Request processing or completion code. So, Sync Scope to the rescue!

But there are some less than wonderful things about using Sync Scope as well. The most obvious downside is that it doesn't allow **any part** of the applicable callbacks to run in parallel within their Sync Scope. Thus, even if the **only** place we need serialization in our EvtIoXxx Event Processing Callbacks is when we update our statistics block (as shown in Figure 1), all the code in our EvtIoXxx Event Processing Callback runs under lock. And the amount of code running under serialization only increases the more we expand Sync Scope through the use of **AutomaticSerialziation**.

For an example, let's look at how using Sync Scope Device affects your driver and device's overall processing. What Sync Scope Device means for your EvtIoXxx functions is that only one of your EvtIoXxx Event Processing callbacks can be running at a time for a given device. Consider the case when your device can have one read **and** one write in progress simultaneously. In this case, you would most likely have two Queues using Sequential Dispatching – one for reads and another for writes. Specifying Sync Scope Device means that while your device can still have two Requests active at the same time, the EvtIoXxx Event Processing Callbacks initiating those Requests could never be running simultaneously. This limits your devices performance and throughput. When you expand your Sync Scope using

(CONTINUED ON PAGE 26)

Understanding Sync Scope... (Cont.)

(CONTINUED FROM PAGE 25)

AutomaticSerialization to include your DpcForIsr, your driver can now only be initiating a single I/O request (a read or a write) **or** completing a single I/O request (of either type) at one time, regardless of how many processor cores there are in the system or what those cores are doing. Consider the possible impact of such a scheme to a driver for a Programmed I/O type device: All the time your driver is moving data to or from your device (using WRITE_REGISTER_BUFFER_UCHAR, for example) you're holding the lock, and your driver is prevented from initiating or completing any other Requests for that device. Not good if you have a device where throughput or maximizing device utilization or reducing Request processing latency is important. Not good at all.

Of course, if your device is slow and your driver doesn't do much to initiate or complete Requests, Sync Scope Device might not be such a terrible thing no matter how many Requests it can have in progress at the same time.

Performance Isn't the Only Concern

Unfortunately, potential lack of performance and lower device utilization due to over-serialization aren't the only issues with Sync Scope. The real complexity of using Sync Scope has to do with IRQL.

As we've discussed previously, the primary feature of Sync Scope is that the Framework automatically acquires and releases a lock around its calls to the applicable callbacks in your driver. But another important feature of Sync Scope is that WDF automatically selects the appropriate type of lock to use to guard those callbacks. And it's this second feature, selecting the appropriate lock type, which can become complicated and cause unexpected consequences.

Recall that by default WDF is allowed to call your driver's EvtIoXxx Event Processing Callbacks at any IRQL <= DISPATCH_LEVEL. As a result, when you select a Sync Scope, WDF defaults to selecting a spin lock as the type of lock it will use to serialize your callbacks.

Now, some of you might be jumping to conclusions and saying "A spin lock! Oh wow... that's a pretty heavy-weight lock to use!" But, in fact spin locks are pretty fast to acquire and release. Consider that acquiring almost any other type of lock involves acquiring and releasing a spin lock for internal serialization purposes. Compared to most other locks, you could say that spin locks are "high speed, low drag." However, there **are** two particular issues that using a spin lock creates.

The first issue is that the code running while a spin lock is held (that is, between the acquire and release) runs at IRQL DISPATCH_LEVEL. Code running at this IRQL is not subject to preemption. This means that, even in the face of quantum exhaustion, a thread running at IRQL DISPATCH_LEVEL will continue running until the IRQL is lowered. This is as it should be, because – by design – code sequences running at IRQL DISPATCH_LEVEL should be short. How short? According to the WDK docs, "a typical DPC routine should run for no more than 100 microseconds" (from the doc page for **KeQueryDpcWatchdogInformation**). This is because DPC routines run at IRQL DISPATCH_LEVEL – no other reason.

Thus when you use Sync Scope and WDF chooses a spin lock, depending on the processing you need to do in your EvtIoXxx routine your driver might wind up spending quite a bit of time at IRQL DISPATCH_LEVEL. And while routines like your DpcForIsr always architecturally **must** run at IRQL DISPATCH_LEVEL, your EvtIoXxx Event Processing Callbacks only **might** run at IRQL DISPATCH_LEVEL. If you set Sync Scope, by default your EvtIoXxx routines will **always** run at IRQL DISPATCH_LEVEL. We'll talk about when and how you might choose to override the defaults later in this article.

Note that running at IRQL DISPATCH_LEVEL won't by itself necessarily hurt the throughput of your driver (assuming you don't count the throughput and device utilization losses inherent in running under lock that we discussed earlier). However, spending an excessive amount of time at IRQL DISPATCH_LEVEL can negatively impact the performance of the overall system. When running at DISPATCH_LEVEL you're basically preventing the Windows scheduler from doing its job. You're monopolizing a core for your use, as your DISPATCH_LEVEL code continues running until it's done.

Keeping It Consistent

The second issue that using a spin lock for Sync Scope serialization causes, involves expanding your serialization domain using **AutomaticSerialization**. You must be sure that your chosen Sync Scope, plus your choices of routines to which you want to

(CONTINUED ON PAGE 27)

Understanding Sync Scope... (Cont.)

(CONTINUED FROM PAGE 26)

extend that Sync Scope with **AutomaticSerialization**, plus the IRQL that the Framework's choice of locking implies, are all consistent and proper. What do I mean? Well, take a look at **Table 1** for starters.

Routine to Serialize	IRQL Requirement
EvtIoXxx	<= DISPATCH_LEVEL
EvtRequestCancel	<= DISPATCH_LEVEL
EvtDeviceFileCreate, EvtFileCleanup, EvtFileClose (only available for Sync Scope Device)	PASSIVE_LEVEL
EvtTimerFunction (if AutomaticSerialization is chosen)	<= DISPATCH_LEVEL (PASSIVE_LEVEL must be specifically configured if desired when Timer is created)
EvtWorkItemFunc	PASSIVE_LEVEL
DpcForIsr	DISPATCH_LEVEL

Table 1 -- IRQL Constraints of Serializable Callbacks

Table 1 shows the various callbacks that Sync Scope can serialize and the IRQL requirements imposed by those callbacks. A careful look at this table will show that there is no single type of lock that can be used that will successfully serialize all potential callbacks. For example, if you let the Framework choose the default spin lock, that will work for your EvtIoXxx Event Processing Callbacks but that **cannot** work for File Object related callbacks or your Work Item callbacks. This is because acquiring a spin lock raises the IRQL to DISPATCH_LEVEL, and the File Object and Work Item related callbacks all must run at IRQL PASSIVE_LEVEL.

Changing the Default Framework Lock Selection

So what can you do if you want to use Sync Scope and (for example) you need to extend that Sync Scope to your File Object or Work Item related callbacks? The answer is that you need to influence the Framework to choose a lock type that's compatible with the callbacks that you want to serialize.

The Framework choses the type of lock type to use based on the constraints imposed on your EvtIoXxx Event Processing Callbacks. As we've now said multiple times, these callbacks by default can run at IRQL DISPATCH_LEVEL or below, and therefore the Framework meets this constraint by choosing a spin lock to implement Sync Scope. You may already be aware that you can constrain your EvtIoXxx Event Processing Callbacks to run at IRQL PASSIVE_LEVEL by specifying an Execution Level constraint when

(CONTINUED ON PAGE 28)

Understanding Sync Scope... (Cont.)

(CONTINUED FROM PAGE 27)

you create your WDFDEVICE. Execution Level is a constraint that is specified in the WDF_OBJECT_ATTRIBUTES structure you specify when your WDFDEVICE is being created. You specify an Execution Level constraint by filling in the **ExecutionLevel** field as shown in **Figure 6**.

```
// ... beginning part of EvtDriverDeviceAdd ...

    WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE(&attributes,
                                            MY_DEVICE_CONTEXT);

//
// Ensure that all our EvtIoXxx callbacks never run in parallel
//
attributes.SynchronizationScope = WdfSynchronizationScopeDevice;
attributes.ExecutionLevel = WdfExecutionLevelPassive;

status = WdfDeviceCreate(&DeviceInit, &attributes, &device);

if (!NT_SUCCESS(status))
{

    // ... whatever ...
    goto Exit;
}

// ... rest of function ...
```

Figure 6 – Constraining EvtIoXxx to be called at IRQL PASSIVE_LEVEL

You can see in Figure 6 we've set **ExecutionLevel** to **WdfExecutionLevelPassive**. This will cause the Framework to always call your EvtIoXxx routines at IRQL PASSIVE_LEVEL.

When you set a PASSIVE_LEVEL constraint on the Execution Level of your EvIoXxx functions **and** you specify Sync Scope, the Framework will take note of your specified Execution Level constraint and use a Mutex to implement Sync Scope. Thus, before calling any of the routines within your Sync Scope, the Framework will acquire the Mutex. And when that routine exists, the Framework will release the Mutex. A Mutex is a type of Windows Dispatcher Object. As a result, any code that runs after the Mutex is acquired and before the Mutex is released runs at IRQL PASSIVE_LEVEL.

Thus, if you specify a PASSIVE_LEVEL constraint on your EvtIoXxx functions, and you specify a Sync Scope, you can now extend that Sync Scope to any of the routines shown in Table 1 that can be called at IRQL PASSIVE_LEVEL. Of course, the problem is now you can't extend your Sync Scope (through the use of Automatic Serialization) to your DpcForIsr, because that must run at IRQL DISPATCH_LEVEL!

Sigh. No size fits all, in this case.

But is That What You WANT?

Specifying IRQL PASSIVE_LEVEL as an Execution Level contraint on your EvtIoXxx functions has another effect, of course. And that effect is that any Requests that arrive at your driver at IRQL DISPATCH_LEVEL (such as those sent by another driver) will have to be deferred until the Framework calls the appropriate EvtIoXxx function in your driver at IRQL PASSIVE_LEVEL. The way the Framework does this is it passes the Request to an internal work item that runs at IRQL PASSIVE_LEVEL and calls the appropriate EvtIoXxx callback in your driver.

And thus we have yet one more item that the use of Sync Scope can impact. If **all you want to do** is extend Sync Scope to include Work Items, you have to set an IRQL PASSIVE_LEVEL constraint on the Execution Level of your EvtIoXxx routines. This, in turn, results in the framework having to defer the processing of any Requests that arrive at IRQL DISPATCH_LEVEL to a work item (instead of having them call directly into your EvtIoXxx callback). If you're a device driver, this also means that if you call **WdfRequestComplete** from your DpcForIsr, and doing so can trigger the Framework to present your driver another Request, the Framework will be forced to use a work item to call your EvtIoXxx Event Processing Callback instead of possibly calling you directly. Neither of these things will be a problem **if** most or all of your Requests arrive directly from user-mode and/or you complete most Requests at IRQL DISPATCH_LEVEL. In both of these cases your driver will already be running at IRQL PASSIVE_LEVEL, and so the Framework need not do any additional processing. But what if you're writing an intermediate driver, and you get most of your work from other drivers in the system? Or if you're a device driver that completes the majority of your Requests in your DpcForIsr? Forcing an IRQL PASSIVE_LEVEL contraint can introduce additional overhead and latency that you don't want.

The Answer Is…

Hopefully, you can now see that while Sync Scope might initially look like a simple solution to a complex problem, it can actually be quite complex when you look at the details. So, what are the alternatives?

(CONTINUED ON PAGE 29)

Understanding Sync Scope... (Cont.)

(CONTINUED FROM PAGE 28)

The main alternative to using Sync Scope to guard various data structures is to do manual serialization. This means that you manually choose a lock, and ensure you acquire and release it properly as required. As previously mentioned, with a little practice this will become almost second-nature to you as a developer. The "trick" (if you can even call it that) is to **only** hold the lock during the specific code sequences in which you manipulate the guarded data structures.

The topic of chosing and using serialization primitives in Windows drivers requires its own article, if not its own book. We'll write a much more detailed article about serialization in a future issue of *The NT Insider*. However, without going into too much detail or discussing too many options, we can provide a brief glimpse at one of the simplest solutions.

```
typedef struct _MY_DEVICE_CONTEXT {

    WDFDEVICE   DeviceHandle;
...
    //
    // Statistics
    // We keep the following counts/values for our device
    //
    KSPIN_LOCK  StatisticsLock;
    LONG        TotalRequestsProcessed;
    LONG        ReadsProcessed;
    LONGLONG    BytesRead;
    LONG        WritesProcessed;
    LONGLONG    BytesWritten;
    LONG        IoctlsProcessed;
...

} MY_DEVICE_CONTEXT, *PMY_DEVICE_CONTEXT;
```

Figure 7 – Spin lock now added to Device Context

The simplest solution to the example problem shown in Figures 1 and 2, is that we could properly serialize access to the statistics in our driver using a spin lock. To do this, we would add the definition of a lock (we'll use a spin lock) to our Device Context previously shown in Figure 1. This updated version is shown in **Figure 7**.

As you can see in Figure 7, we simply reserve space for a KSPIN_LOCK structure in our Device Context in a field we've named **StatisticsLock**. Note that the spin lock structure needs to be initialized before its first use (probably in EvtDriverDeviceAdd, just after the WDFDEVICE has been created) by calling **KeInitializeSpinLock**.

We would then update the code originally shown in Figure 2 to acquire the spin lock before we update the statistics and release the spin lock immediately afterwards. This is shown in **Figure 8**.

In Figure 8, you can see that after validating and getting the pointer and length of the user data buffer, we acquire the spin lock we previously defined in our Device Context, by calling **KeAcquireSpinLock**. We then manipulate the statistics. When we're done with updating the statistics, we release the spin lock by calling **KeReleaseSpinLock**.

Note that it's entirely safe to use a spin lock here because we know, as a matter of WDF architecture, that the Device Context is always located in non-paged pool. And we know that this code will never run at an IRQL greater than DISPATCH_LEVEL (a specific requirement of using this particular type of spin lock).

```
VOID
MyEvtIoWrite(WDFQUEUE     Queue,
             WDFREQUEST   Request,
             size_t       Length)
{
    NTSTATUS                status = STATUS_SUCCESS;
    WDFDEVICE               device = WdfIoQueueGetDevice(Queue);
    PMY_DEVICE_CONTEXT      context;
    PVOID                   buffer;
    PVOID                   length;
    KIRQL                   oldIrql;

    //
    // Get a pointer to my device context
    //
    context = MyGetContextFromDevice(device);

    //
    // Get the pointer and length to the requestor's data buffer
    //
    status = WdfRequestRetrieveInputBuffer(device,
                                           MY_MAX_LENGTH,
                                           &buffer,
                                           &length);

    if (!NT_SUCCESS(status)) {

        // ... do something ...
    }

    //
    // Update our device statistics
    //
    KeAcquireSpinLock(&context->StatisticsLock,
                      &oldIrql);

    context->TotalRequestsProcessed++;
    context->WritesProcessed++;
            context->BytesWritten += length;

    KeReleaseSpinLock(&context->StatisticsLock,
                      oldIrql);

    //
    // ... rest of function ...


    return;
}
```

Figure 8 – Locking around the statistics manipulation using a spin lock

(CONTINUED ON PAGE 30)

Understanding Sync Scope... (Cont.)

(CONTINUED FROM PAGE 29)

If the only thing you need serialization for in your EvtIoXxx functions is the statistics, using manual serialization would be a far better choice than using Sync Scope.

Sync Scope Guidance

Given everything we've discussed so far, when does Sync Scope make the most sense? Different developers will, of course, have different opinions on this topic. Here at OSR, here are the guidelines we typically recommend:

1. Recognize that Sync Scope is almost never **the best** method of serialization in a WDF driver. Think carefully about your alternatives before specifying a Sync Scope.

2. Sync Scope can be an acceptable method in a production driver if:
   a. A great deal of code in your EvtIoXxx function requires serialization; OR
   b. The amount of time spent within the Sync Scope will be very low over time. By this, we mean that the your EvtIoXxx functions and any other functions within your Sync Scope run infrequently; OR
   c. You're very new to the world of Windows driver development, and making your driver work correctly is more important that making your driver work optimally; OR
   d. Less than optimal performance – for your driver and the system on which your driver is running -- is acceptable.

3. Avoid using Sync Scope as a quick and dirty method of using a "big lock" model during driver bring-up. It's long been our experience that building the proper serialization into a driver while the code is being written is **far** easier than retrofitting proper serialization to that driver afterwards. Ask us how we know this (blush).

4. Sync Scope, extended to whatever functions you might require, can be a quick way to check for a race condition in your EvtIoXxx routines when you're trying to chase-down a driver bug. Of course, this isn't nearly as nifty as it sounds due to the inherent issues in using Sync Scope that we've discussed in this article.

Thus, the bottom line on Sync Scope is that it's probably best if you use it only under some limited and special conditions. You know the saying, "If it sounds too good to be true, it probably is"? That's the bottom line with Sync Scope.

Have fun, and be careful out there.

Follow us!

Peter Pontificates... (Cont.)

(CONTINUED FROM PAGE 5)

So, no. OSR won't be moving to a hosted email solution anytime soon. And we won't be backing up our confidential stuff "to the cloud" either, thank you very much. And that's regardless of the blather that IT consultants propound, the number of official-looking yet useless SAS 70 Certificates presented, or even the sometimes sincere yet predictably unreliable promises made by sales people. What amazes me most is how it's not obvious to everyone that moving these things to the cloud is a Really Bad Idea. I've thought about it, and the only explanation I can come up with is mass delusion.

Aren't you glad you have me to show you the light?

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

Follow us!

OSR Seminar Schedule

Seminar	Dates	Location
WDF Drivers: Core Concepts	22-26 September	Boston/Waltham, MA
Internals & Software Drivers	20-24 October	Dulles/Sterling, VA
Developing File Systems	4-7 November	Seattle, WA
Kernel Debugging & Crash Analysis	10-14 November	Waltham, MA
WDF Drivers: Advanced Implementation Techniques	16-19 March 2015	Palo Alto, CA

OSR Seminars

We "Practice What We Teach" For a Reason

When we say "we practice what we teach", this mantra directly translates into the value we bring to our seminars. But don't take our word for it...below are some results from recent surveys of attendees of OSR seminars:

- "[Instructor] is a gifted teacher. He truly made the concepts **easy to understand and learn**".
- "**Wonderful**, would definitely look into other courses by OSR".
- "[Instructor] was simply awesome. He did a very good job of making the **class room training interesting and interactive**.
- "In a nutshell, the **best seminar** I've ever attended."
- "The OSR staff was eager and willing to help with anything what-so-ever related to the class."
- "[Instructor} was an excellent instructor; he understood the material thoroughly and **presented it clearly**.
- "This was a great seminar with good pacing, useful information and **a high level of detail**".
- "[Instructor] was excellent. He knows the material well and **fielded all questions with ease** and provided great information and feedback. He was personal, engaging and **everything thing a presenter should be**.
- "[Instructor] was **fantastic** and I can easily say that this has been one of the **best training seminars I have attended**".
- "I was **VERY impressed** with the content and the **instructor's knowledge of the subject matter**. All questions were answered for all the students and/or researched quickly, if an answer was not readily available."

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.