ISSUE VOLUME
01/23
JAN-FEB
2017

The NT Insider

A publication of OSR Open Systems Resources, Inc.

```
//
// ReadNewNtInsider
//
// Read this issue of The NT Insider, based on the reader's
// interests.
//
NTSTATUS
ReadNewNtInsider(PACCESS_TOKEN Token)
{
    NTSTATUS status;
    PTOKEN_GROUPS groups = NULL;

    //
    // Retrieve the groups for this reader
    //
    status = SeQueryInformationToken(Token,
                                     TokenGroups,
                                     (PVOID*)&groups);

    if (!NT_SUCCESS(status)) {

        goto leave;
    }

    //
    // Everyone reads Peter Pontificates
    //
    ReadPeterPontificates();

    //
    // All driver devs should read these articles
    //
    if (InsiderIsGroupMember(groups, SECURITY_INSIDER_DRIVER_DEV)) {


        //
        // Best Practices for Windows Driver Developers
        //
        ReadArticle(Page3);

        //
        // Develop and Test Complex Drivers in User Mode
        //
        ReadArticle(Page8);

    }

    //
    // WDF developers should read about device objects and protections
    //
    if (InsiderIsGroupMember(groups, SECURITY_INSIDER_WDF_DEV)) {


        //
        // Making Device Objects Accessible... and SAFE
        //
        ReadArticle(Page6);

    }

    //
    // File system and file system filter devs…
    //
    if ( InsiderIsGroupMember(groups, SECURITY_INSIDER_IFS_DEV) ||
         InsiderIsGroupMember(groups, SECURITY_INSIDER_FILTER_DEV) ) {

        //
        // What's New in Win10 for File Systems and Filters?
        //
        ReadArticle(Page10);
    }

leave:

    if (groups) {

        ExFreePool(groups);
    }

    return(status);
}
```

The NT Insider

Published by
OSR Open Systems Resources, Inc.
105 Route 101A, Suite 19
Amherst, New Hampshire USA 03031
(v) +1.603.595.6500
(f) +1.603.595.6503

http://www.osr.com

Consulting Partner
Peter G. Viscarola

Engineering Partner
Scott Noone

Executive Editor
Daniel D. Root

Contributing Editors
OSR Staff

Send Stuff To Us:
NtInsider@osr.com

Single Issue Price: $15.00

Get Social with OSR

Real -Time Updates

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

WDK Problem Fixed, Plus Preview Access For All
I wanted to update you on the status of a few issues in the world of the Windows Driver Kit, and also let you know about new options for early access to new versions of the WDK.
http://www.osr.com/blog/2016/10/24/wdk-problem-fixed-plus-early-access/

C Is Not Reasonable
Those of you who've read my Pontifications over the years know that the things that annoy me are truly countless in number. But most of the things that annoy me do so because I simply cannot understand why they are the way they are.
http://www.osr.com/blog/2016/09/19/c-reasonable/

Become More Knowledgeable... Instantly!

We email our friends when we've got something interesting to say. Join the list!

Send a blank email to join-osrhints@lists.osr.com and we'll add you to the list. We don't have THAT much to say. You'll probably get one or two emails a month.

Debugging Target RS1? Good Luck!
The hits to driver developers just keep on coming when it comes to RS1.
http://www.osr.com/blog/2016/08/26/debugging-target-rs1-good-luck/

Careful With Your WDK Updates!
I can never help myself when it comes to updating to the latest WDK. It's always exciting to diff the old contents with the new and see what's really going on in the operating system.
http://www.osr.com/blog/2016/08/03/careful-wdk-updates/

Careful With Your VS 2015 Updates!
As welcome as the Windows updates themselves are, they also come with updates to VS, the SDK, and the WDK. In the past 24 hours we've discovered that it pays to be careful in how you apply these updates.
http://www.osr.com/blog/2016/08/03/careful-vs-2015-updates/

WDK 14393 Code Analysis Enforces POOL_NX_OPTIN, Breaks POOL_NX_OPTIN
In an interesting twist, build 14393 of the WDK now enforces the use of NX non-paged pool.
http://www.osr.com/blog/2016/08/02/wdk-14393-code-analysis-enforces-pool_nx_optin-breaks-pool_nx_optin/

Fix WDK Doc Issues—Yourself. And Fast!
If you use the WDK doc set all day, every day, you almost certainly have a few pet peeves — things in the docs that drive you nuts and that you'd like to change. Now, you make those changes yourself! Well, almost.
http://www.osr.com/blog/2016/08/02/fix-wdk-doc-issues-fast/

Driver Signing—More Details Emerge
Microsoft has just published a new Channel 9 Video that explains many of our long-standing questions about driver signing.
http://www.osr.com/blog/2016/06/02/driver-signing-details-emerge/

Follow us!

Best Practices for Windows Driver Developers

Best Practice: ***A procedure that has been shown by research and experience to produce optimal results and that is established or proposed as a standard suitable for widespread adoption.*** (Merriam-Webster)

We spend a lot of time here at OSR, both in our offices and in our classes, discussing what we believe to be Best Practices for Windows driver development. We thought it would be a good idea to try to enumerate these best practices. We hope to return to this list, updating it frequently, and make it a regular item here in *The NT Insider*.

We realize it won't be possible for every project to follow every one of these suggestions. That's fine. These are meant to be aspirational guidelines. Sometimes it will make the most sense – and even be a best practice – to *not* follow some of the listed practices, depending on the details of your project. Remember, these are guidelines. They're not meant to substitute for your good engineering judgement.

Tool Chain

Use the most up-to-date version of the WDK, and of Visual Studio supported by that WDK.
Why: Every version of the WDK brings new features, and includes fixes from previous versions.

Driver Model

When possible, use the most modern driver model that applies to your project. For example, use WDF (KMDF or UMDF) instead of WDM. Use the file system Mini-Filter model, instead of creating yet another legacy file system filter.
Why: In this case, newer really *is* better. The newer driver models will help you avoid lots of problems that some of the older models entail.

Coding/Building

Even if you write in C, use the C++ compiler.
Why: It's worth it to use the C++ compiler, even if you just use the strong type-checking alone. Later, you may find other modern features of the C++ language that make sense to use and that are compatible with the C-language interfaces provided by Windows.

Use the role-type annotations for your drivers, where provided. By these, we mean the declarations such as EVT_WDF_DRIVER_DEVICE_ADD in WDF, or DRIVER_INITIALIZE in WDM.
Why: These annotations provide additional useful information to both Code Analysis and Static Driver Verifier. They help those tools better understand your code.

Use SAL Annotations for your internal driver functions – At Least for locking and IRQL checking.
Why: Concurrency problems, and IRQL violations, are some of the most common – and the most difficult to diagnose – problems that people have in writing Windows drivers. We have found that using annotations for at least these items can be invaluable in identifying bugs early.

(CONTINUED ON PAGE 35)

Peter Pontificates

Yes, It's Windows 10...But *Which* Version?

Maybe I've just been doing this too long. Or maybe, just maybe, the topic has become just too confusing for any reasonable human being to comprehend.

What topic is it that has me so befuddled? Windows version numbers. And releases.

What gives me hope that I'm not the only one who's close to terminally lost is that regularly – like every day – I have conversations like the following:

Tester: It crashed. Again. Your driver blue screened.
Me: OK… Which OS version were you running?
Tester: Windows 10.
Me: Yes, but what *version* of Windows 10?
Tester: I thought Windows 10 *was* the version.
Me: Well, Yes. Er, no. Er… Windows 10 is the… I dunno. The name.
Tester: The system was sitting there and it may have done some updates.
Me: All the updates? Were you running RS1?
Tester: What's RS1?
Me: Redstone. Redstone 1. The latest release. Not counting pre-releases or the fast ring or anything.
Tester: Redstone? You mean, like the arsenal?
Me: No, ah…
Tester: Like Sumner?
Me: No! More like from Minecraft. Why the fuck are we talking about this!? *Which version of Windows were you running when my driver failed*?
Tester: OK, OK… I don't know what version. If 10 isn't the version, I don't know what version. But the properties screen says build 10586.
Me: Ah! V1511.
Tester: So, is that Minecraft?
Me: No, it's Threshold. Threshold 2. I think.
Tester: Threshold isn't Minecraft? Is it Redstone?
Me: Stop with the Minecraft! Please! Threshold is an older release of Windows 10. Before Redstone. It's from… ah… I don't remember.
Tester: RTM?
Me: No. Look, either go back to the lab now or I'll lock you in the engineering bathroom.

Now, to be sure, the above conversation casts me in a heroic role, because in that conversation I can actually associate build 10586 with Version 1511 and the codename Threshold. Which in real life I may or may not be able to do on any given day, without having to resort to the list on my whiteboard.

Back in the day, it was enough to have honest build numbers and straight forward OS version numbers. I remember NT V4, which was build 1381. Build numbers were strictly increasing each day (pretty much). Build 1381 got that number because it was the thirteen hundred and eighty first time the OS was built by the Build Lab. And Build 1381 got RTM'ed as NT V4, because after 1381 builds, the code was "ready to be released."

Somewhere along the way, build numbers got corrupted by rounding them up to arbitrary values at RTM. And about that same time, we stopped getting ordinary OS version numbers and started getting OS release names. That's how we got Windows XP, which was build 2600 (nice, round, fictitious, build number), and which was also V5.1.

(CONTINUED ON PAGE 5)

Peter Pontificates... (Cont.)

(CONTINUED FROM PAGE 4)

Of course, there have always been internal code names as well. NT V4 was SUR (Shell Update Release -- I had to Google to remember that). And Windows XP was Whistler. In Windows 10 times we've had Threshold and Redstone.

And for the past few releases, version numbers have become pretty much honest again. It's not unreasonable to think that build 10586 (Win10, TH2) is 346 builds newer than 10240 (TH1, Win10 RTM) that preceded it. It's a bit more complicated than that in reality, but… you know.. close enough.

But while build numbers have gotten more honest, the version numbers themselves have gotten out of control. For example, let's take the most recently released version of Windows:

Name: Windows 10 Anniversary Update
Codename: Redstone 1 (RS1).
Build: 14393
Version: Can't say … It depends on what you mean

Yup. Defining the version number just isn't that straight forward.

You would think that Windows 10 would be easy. Because in the name "Windows 10" "10" is the version number, right? Well, not necessarily. Remember Windows 7? That wasn't Windows V7.0, it was Windows V6.1. But is Windows 10 actually Windows V10.0? Well, yes! At least, it is *sometimes*. It actually depends on who asks, and how they ask. If an application running on Windows 10 checks the version of the OS by calling **GetVersionEx**, it will either get back 10.0 or 6.2, depending on how the application is manifested. Or something.

Not to mention, there are actually multiple versions of Windows 10, that (for properly manifested applications) all return V10.0 – There's Windows 10 RTM (AKA TH1, build 10240), there's Windows 10 November Update, which is V1511 (AKA TH2, build 10586), and there is Windows 10 Anniversary Update (AKA Redstone 1, build 14393).

And don't even get me started on which KMDF and UMDF versions are available on each version of Windows. Or fast ring and slow ring. My head hurts enough already.

Follow us!

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

KERNEL DEBUGGING & CRASH ANALYSIS SEMINAR
I Tried !analyze-v...Now What?

You've seen our articles where we delve into analyses of various crash dumps or system hangs to determine root cause. Want to learn the tools and techniques yourself?

"The instructor exhibited a very comprehensive knowledge of the material, added with an incredible ease in explaining a very complex subject. I highly recommend this course."
- Feedback from an attendee of THIS seminar

Upcoming presentation: Amherst, NH (OSR) 26-30 June

Names, Security Descriptors and Device Classes
Making Device Objects Accessible...and Safe

Over the years, much has been written – both in the MSDN documentation and here in *The NT Insider* – about how the Device Objects a driver creates can be accessed. And since Windows 2000 changed the world by introducing the concept of PDOs and FDOs, much has also been written about Device Object security. Alas, much of what's been written has been wrong, misleading, outdated, confusing, confused, or some combination of all of these.

When we started writing this article, we figured it would be straight-forward. You know, crank something out for The NT Insider about a topic that is helpful and about which we have a lot of experience. Making Device Objects secure and accessible: How hard can that be? Well, surprise! Once we started a careful analysis of each possible option, and the interaction of those options, we discovered the proverbial "can of worms." It turns out to be shockingly easy to screw things up. And it's scary simple to end up with a different protection on the Device Objects in your device stack than you expected.

In this article, we'll try to address some of the most basic questions about how a WDF function driver can securely make its devices accessible. Our goal is to define and describe what we believe to be best practices for WDF (specifically, KMDF) drivers. We're going to ignore WDM drivers because, well, you probably shouldn't be writing WDM drivers these days. And if you are, you should already know how to deal with security.

Quick Review: PDOs and FDOs

Let's start at the beginning, with PDOs and FDOs, because this is actually where most of the trouble starts.

As a rule, every device in Windows is discovered through the Plug-and-Play (PnP) process. The *only* exceptions to this rule are (a) software-only drivers that we refer to as "kernel services", and (b) super-ancient hardware drivers that use the original Windows NT model. "Kernel services" are sometimes also referred to as "legacy style software drivers" or "NT V4 style software drivers." The drivers in these exception categories create their Device Objects within their **DriverEntry** entry point. Lots of folks (including us) write kernel services to do things like monitor process creation, watch for registry changes, or provide other sorts of services from kernel mode. For the purposes of this article, we're going to ignore all drivers that aren't started by PnP.

So… as we said… as a rule, every device in Windows is discovered through the PnP process. This is true regardless of whether the device lives on a dynamically enumerable bus (such as PCIe or USB) or on a bus where the attached devices can't be discovered at run time (such as I2C or SPI). When a bus driver enumerates a device on its bus, it creates a Device Object that represents the *physical* instance of the discovered device on its bus. In WDF, the bus driver creates this Device Object using the function **WdfDeviceCreate**. This Device Object, created by the bus driver, is referred to as a Physical Device Object, or PDO for short.

As part of the overall PnP process, the PnP Manager queries the bus driver for a list of its "child devices." If the bus driver has discovered any devices on its bus, it replies with a list of pointers to the PDOs that is has previously created to represent those devices.

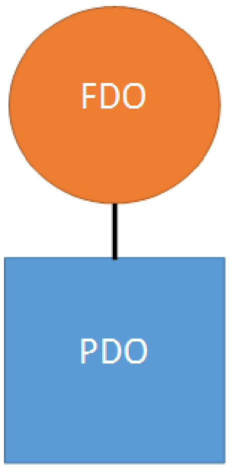


Figure 1—Basic Device Stack

For each PDO returned from a bus driver (with some limited, special-case, exceptions), Windows attempts to find and start a driver that will be responsible for the functional aspects of the device. This is the function driver. It's at this point that WDF function drivers are called at their EvtDriverDeviceAdd Event Processing Callback. Within this Callback, a WDF function driver creates the Device Object (using **WdfDeviceCreate**) that represents the *functional* aspect of the device. This Device Object, created by the function driver, is referred to as the Functional Device Object or FDO for short.

After the FDO has been created, it is attached to the underlying PDO that was previously created by the bus driver. The result is a pair of Device Objects that together represent (a) the physical presence of a device on a given bus (PDO), and (b) the functional aspect of that device (the FDO). This pair forms the basis of the Device Stack, and is shown in *Figure 1*. For the sake of simplicity, we're ignoring filter drivers and their Device Objects in this discussion.

It's important to note that, regardless of how a device is accessed, any I/O operations that are sent to a device *will always enter at the top of the Device Stack* in which the device appears. In other words, I/O requests will always go to the FDO first. This makes sense, because it's the function driver (the one that created the FDO)

(CONTINUED ON PAGE 7)

Making Device Objects Accessible... (Cont.)

(CONTINUED FROM PAGE 6)

that is responsible for the functional aspects of the device. And it's most typically only the function driver (and not the bus driver) that knows how to process I/O requests for the device. In fact, the bus driver is rarely involved in processing typical I/O operations.

The fact that in Windows we have *two Device Objects* that together represent *one single device* is the root cause of many of the problems and much of the confusion about how devices are accessed and protected. When you add to this the fact that there are multiple ways that you can access the device associated with these Device Objects, things can get tricky, fast.

Device Object Attributes

Whenever a Device Object is created, the creator can optionally specify a wide variety of characteristics and attributes. For the purposes of this article, the most interesting among these are device name, Device Setup Class, and default Device Object security.

Device Naming

Specifying a device name provides an "internal" name, otherwise known as a "native" name, for the Device Object. This name is not easily accessible from user mode, but rather is typically used by other kernel mode entities to find or identify the device by name.

Naming an FDO is strictly optional, and best practice dictates that you should not name your FDO unless you absolutely need to. We'll have more to say about this later in this article, when we examine the implications of naming your Device Objects in more detail.

Due to some dubious architectural choices in Windows 2000, PDOs must always be named. However, because the bus driver rarely intends this name to be used to access the device, the name can be, and almost always is, arbitrary and meaningless (such as "\Device\NTPNP_0005"). There's even a commonly used option to have the I/O Manager automatically generate the device name for a PDO.

In KMDF, if you want to provide a name for the Device Object you're creating (regardless of whether that Device Object is a PDO or an FDO), you use a method on the **WDFDEVICE_INIT** structure. This method is **WdfDeviceInitAssignName**, as shown in the following example:

```
DECLARE_CONST_UNICODE_STRING(InternalName, L"\\Device\\MissileLauncher1") ;

status = WdfDeviceInitAssignName(DeviceInit,
                                 &InternalName);
```

The code above assigns the internal (native) name "MissileLauncher1" to the Device Object that will be created with the **WDFDEVICE_INIT** pointed to by *DeviceInit*. Note that, by convention, the name is placed in the Object Manager's \Device\ directory.

(CONTINUED ON PAGE 24)

Time-Saving Strategies
Develop and Test Complex Driver Code in User Mode

You can think of just about any driver performing three separate types of processing:

1. Receiving and managing Requests from the OS.
2. Creating and managing internal state and data structures.
3. Interacting with other components in the system – hardware, other drivers, or other random pieces of the OS – to complete the processing of Requests that have been received.

For certain classes of drivers, item number 2 – the creation and management of the driver's internal state and data structures – represents the majority of the code and complexity in the project. Developing and testing such drivers can pose unique challenges due to the amount of code and the complexity of the algorithms involved.

Over the past few years, a new and effective strategy has evolved for more easily and efficiently developing and testing these types of drivers. That strategy is to *develop and test the code in user mode first*. Then, after the code is proven to be solid, move it to kernel mode.

Here at OSR, we've used this approach in our own development projects, and we're aware of it being used for complex driver projects at Microsoft as well. Universally, people start out being uber-skeptical of the value of this approach: Won't it be a yuuuge PITA to create the infrastructure necessary to write and test my driver code in user mode? Will I really save enough time by doing this to make it worthwhile? Won't I wind-up re-writing a lot of code when I eventually move to kernel mode? Can't I just get on with writing my kernel mode code and get my project done?

In every case I've heard of or been involved with, the effort required to develop and test in user mode has been well worthwhile. Developing and testing complex code in user mode has actually *saved* tons of time, not cost time as you might guess. And, more importantly, the quality that's resulted has been much better than would likely have been possible in the same amount of time if development and testing had taken place strictly in kernel mode.

Not convinced? I don't blame you. I was skeptical at first as well. In this article, based on experiences in a few recent projects we've done, I'll provide a number of hints and tips that should help you succeed if you decide to give this technique a try.

Some Recent Experience

As a real-life example where the user mode development and testing strategy was helpful, we'll use a driver we wrote recently here at OSR. This driver required a caching package that temporarily stored disk writes in memory. We initially wrote and tested this caching package entirely in user mode.

The caching package allocates an appropriately sized memory buffer, stores the data being written disk into that buffer, keeps track of the disk sector range to which that (now cached) data was to have been written, and then completes the write operation without ever sending it to disk. When a disk read occurs, the caching package checks to see if it data for all or part of the sought range is in cache. If any of the sought sectors are in cache, the cache package satisfies the disk read using the cached data (plus any data that was not cached, read directly from disk). While there were a few other wrinkles in the real project, those are the functional requirements at their most basic.

(CONTINUED ON PAGE 9)

Time Saving Strategies... (Cont.)

(CONTINUED FROM PAGE 8)

This project was a perfect choice for user mode development and testing, because the major complexity in the driver was the caching code. The algorithms to track which blocks were stored in cache, locate the buffers associated with the blocks, allocate new cache storage buffers and their associated tracking structures if necessary, and to allow blocks of stored data to be read and written comprised almost all of the driver's complexity. Both data structures and locking strategies had to be designed to ensure as many things as possible could happen in parallel (for performance reasons) while also ensuring the integrity of data structures and data.

Any good design would surely isolate the caching code from the Request processing code. And, of course, that's how this driver was designed. This design made developing and testing the caching code in user mode a pretty easy task. The task was made easier still by the fact that Request processing code had dependencies on the caching code, but there were no dependencies from the caching code on the Request processing code. Thus, it was easy to treat the caching code as a stand-alone entity. By the way, there is a moral to this story: A good design, with good isolation of functional components, can go a long way toward facilitating user mode development and testing.

With that background, let's look at a few hints and tips that might help you adopt this strategy for your own use.

Write a Regular Program, Not a User-Mode Driver

When I say, "develop and test your driver code in user mode", I'm *not* suggesting that you should try writing your driver using UMDF. Now, let me hasten to add that UMDF is definitely a good thing. And it is certainly true that writing and testing a driver using UMDF for certain classes of devices can definitely be quicker and easier than writing the same driver using KMDF. But for the class of driver we're focusing on here – drivers with lots of complex internal processing and data structures – I'm not suggesting you start with UMDF. I'm actually suggesting you code-up and test all the complex processing and data structure management as an ordinary user mode program from within Visual Studio.

The primary thing that writing your code as an ordinary user mode program achieves is *dramatically* faster "cycle time." Think about it. When you want to make a change to the code in your kernel mode driver, you need to edit the code, stop the device, copy the executable, and restart the device. If there was a crash involved (or your driver needs to start at boot time) you'll need to reboot the system. You need to fight with WinDbg when it doesn't handle your breakpoints properly or when your client and target don't sync-up properly. This all takes a lot longer than making a change and restarting your program under the Visual Studio debugger. In my experience, the time you save *really* adds up. You simply have to experience it to realize how dramatic the difference is.

It's important to understand that the code you want to focus on in your user mode development and testing is *not* the code that directly performs device initialization, or directly receives Requests and processes them. Rather, the code you want to focus on testing is your algorithmically complex code. So, in the case of the example disk caching driver we described previously, the only code we wrote and tested in user mode was the disk caching code. Code to get Requests and process them, and code to deal with things like power state changes, was left for standard kernel mode development and testing.

Throw-Away or Sustainable Infrastructure?

When you decide to code and test parts of your driver in user mode, perhaps the biggest decision you'll face is whether you want the user-mode infrastructure that you create to be sustainable after your initial development and testing process has ended. Creating a "throw away" infrastructure in user mode is easy. Creating a sustainable infrastructure that you can use for modifying and testing later versions of your driver code is harder, but brings additional advantages.

Restricting the use of user mode to only the initial design and development of your driver is the simpler alternative. Because the infrastructure you're creating isn't permanent, you're pretty much free to hack and whack whatever you need. No need to be tidy. No need to write comments that are comprehensible by third parties. No need to be embarrassed by writing ugly code. All you need to do is to make something work temporarily, while your algorithms and data structures are being developed and validated.

(CONTINUED ON PAGE 20)

Windows 10: What's New for FS/Filters?

File System Related Changes in Redstone 1

In July 2016, Microsoft released the latest major update to Windows 10. There are various names for this release, but Redstone 1 or RS1 for short, seems to be the one that most of the technical community uses. The release is more officially called Anniversary Edition, and has been designated version 1607. The changes in RS1 are also present in Windows Server 2016 (S16) as in many cases the binaries are identical between the systems.

To keep you up to speed, we'll highlight a selected a set of changes that are likely to be of interest to people working with Windows file systems, including those developing file system mini-filter drivers.

The process we used to identify these changes was a systematic comparison of the header files (ntifs.h, ntddk.h, wdm.h, and fltkd.h), followed by some select careful observation of a running RS1 system. While many of these changes will impact file system related drivers including mini-filters, some may impact other drivers as well. If you still have a legacy file system filter driver, you'll almost certainly want to be aware of these changes so you can accommodate them as you migrate to the mini-filter model.

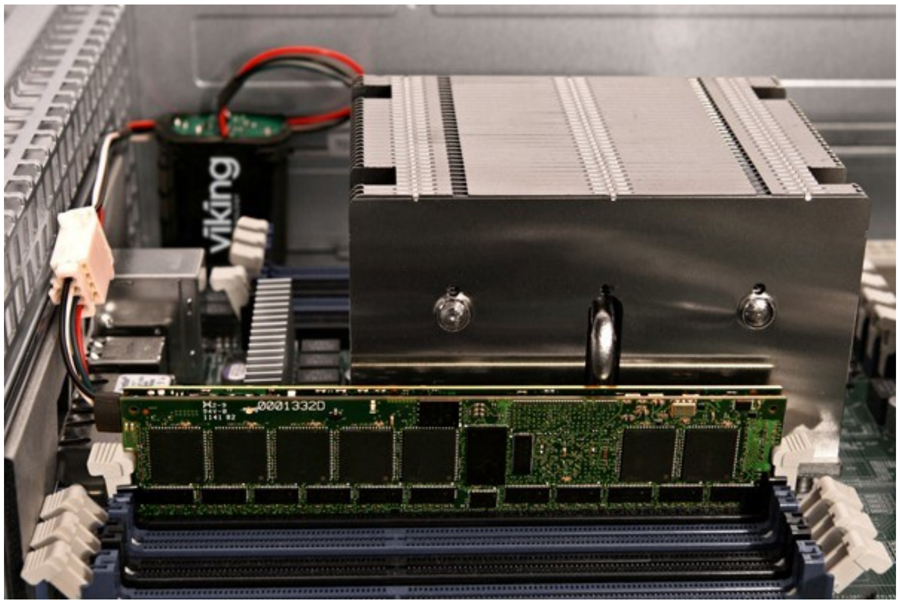


Figure 1 –Persistent Memory
(courtesy Viking Technologies, used with permission)

Direct Access Memory Device Support

Windows 10 (and Server 2016) now include support for persistent memory storage devices. These NVRAM based devices use normal memory slots, but provide persistent storage, which can be used by a file system in order to obviate the need to do any RAM-based caching, due to the performance of the device itself.

Support for these new persistent memory devices has been present in Linux for various file systems and is now supported in Windows 10. For those interested in the device driver aspects of this new technology, there are two new drivers:

- A Storage Class Memory bus driver (scmbus.sys)
- An SCM disk driver (scmdisk0101.sys)

SCM devices operate in one of two modes: Block Mode, or Direct Access Storage (DAS) Mode. In Windows, the mode is chosen when the SCM device is formatted. In Block Mode, SCM devices appear as "ordinary" storage volumes and thus maintain all existing storage semantics. This provides perfect application compatibility, but requires I/O operations to traverse (a slightly optimized path through) the Windows storage stack.

(CONTINUED ON PAGE 11)

File System Related Changes...(Cont.)

(CONTINUED FROM PAGE 10)

DAS Mode is much more interesting. It is supported in RS1 by the NTFS and ReFS file systems. The key benefit for applications using file systems that support the DAS Mode interface is it provides **zero copy** access. Memory mapping of the file directly maps the SCM memory into the address space, whether it is an application or the Cache Manager.

There are some behavior changes with the introduction of SCM in SAS Mode as well:

- Potentially different types of storage failure
- NTFS: no encryption, compression or TxF (transaction) support
- ReFS: no integrity streams, no cluster bands, no block cloning
- No Bitlocker support
- No volume snapshots
- No mirrored or parity support (Storage Spaces or Dynamic Volumes)
- Modification Time and USN Journal semantics are altered slightly (“last update” is the date of memory mapping)
- Directory Change Notification occurs at memory mapping time

Some file system filter drivers, notably data transformation filters (encryption/compression/HSM), may be impacted by these changes. Filters must explicitly indicate if they support direct access storage (by setting the **FLTFL_REGISTRATION_SUPPORT_DAX_VOLUME** bit in their registration structure **Flags** field). Otherwise, the filter cannot attach to SCM volumes.

Driver Level Changes

The **DO_DAX_VOLUME** bit is set in the device object of an SCM device (this is defined in wdm.h, ntddk.h and ntifs.h):

```
// DO_DAX_VOLUME - If set, this is a DAX volume i.e. the volume supports mapping a file directly
// on the persistent memory device.  The cached and memory mapped IO to user files wouldn't
// generate paging IO.
//
#define DO_DAX_VOLUME           0x10000000
```

Filesystems that support DAX, should indicate this in their file system attributes (defined in ntifs.h):

```
//
//  When enabled this attribute implies that the volume supports byte addressable
//  mode.  A mode where reads / writes on mapped files happen directly on the
//  storage device, without going through the file system and the storage stack.
//
//  NOTE: This attribute only mean that the file system supports.  It doesn't
//  imply that the storage hardware is capable.  The storage hardware should be
//  a byte addressable persistent memory device, to let one map files directly
//  on the storage device.
//
#define FILE_DAX_VOLUME       0x20000000   // winnt
```

A file system (and filter driver) can test to see if a volume is a DAX volume by using the new **FsRtl** routine for this purpose (ntifs.h):

```
BOOLEAN
FsRtlIsDaxVolume (

  _In_ PFILE_OBJECT FileObject

    );
```

(CONTINUED ON PAGE 12)

File System Related Changes...(Cont.)

(CONTINUED FROM PAGE 11)

Because SCM type devices are memory, and memory is typically addressable in units smaller than the size of a sector, using SCM can introduce new failure modes to applications. For example, in some SCM type devices a write operation could be interrupted mid-sector due to a system crash or power failure. There is a new I/O stack location bit (defined in wdm.h) to deal with this problem:

```
#define SL_PERSISTENT_MEMORY_FIXED_MAPPING 0x20    // valid only with persistent memory device and IRP_MJ_WRITE
```

This bit is optional, but when set indicates the SCM device is using Intel's block translation table mechanism (defined in the NVDIMM Namespace Specification) that guarantees sector level atomic writes. This feature provides more compatibility with the way traditional disks fail and thus minimizes the impact of unexpected types of failures when using SCM type devices.

In order to accommodate this new feature for NTFS and ReFS, there is a new Cache Manager routine for initializing the cache for persistent memory devices (defined in ntifs.h):

```
NTKERNELAPI

VOID

CcInitializeCacheMapEx (

  _In_  PFILE_OBJECT FileObject,

  _In_ PCC_FILE_SIZES FileSizes,

  _In_ BOOLEAN PinAccess,

  _In_ PCACHE_MANAGER_CALLBACKS Callbacks,

  _In_ PVOID LazyWriteContext,

  _In_ ULONG Flags

    );
```

Naturally, if your file system will support this feature, you will need to use this routine to support direct access storage as well.

The relevant new Cache Manager flag (defined in ntifs.h):

```
//
// The following flags are valid Flags parameter that CcInitializeCacheMapEx accepts
//

#define CACHE_USE_DIRECT_ACCESS_MAPPING         (0x00000001)
```

(CONTINUED ON PAGE 13)

File System Related Changes...(Cont.)

(CONTINUED FROM PAGE 12)

File system filter drivers attached to such devices need to understand that these *do not* behave like normal file systems. For example, the normal pattern of Paging I/O is different than your filter might be familiar handling: specifically, persistent memory devices are directly accessed for cached I/O and thus do not cause any paging I/O activity.

Of course, this is just a brief overview of SCM devices on Windows. There's lots more to know about these new devices, that promise the possibility of a major shift in how certain data is stored on Windows systems. We'll write more about SCM devices in a future issue of *The NT Insider*.

Reparse Point Changes

In the past, NTFS has required that directories be empty prior to applying a reparse point. Redstone now introduces support for reparse points on *non-empty* directories. Note that not all reparse points support this feature: it is a characteristic of the specific reparse point value.

Microsoft has introduced a bit in the reparse point tag that will be used moving forward (from ntifs.h):

```
//
// The reparse tags are a ULONG. The 32 bits are laid out as follows:
//
//   3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
//   1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
//  +-+-+-+-+-----------------------+-------------------------------+
//  |M|R|N|D|     Reserved bits     |       Reparse Tag Value       |
//  +-+-+-+-+-----------------------+-------------------------------+
//
// M is the Microsoft bit. When set to 1, it denotes a tag owned by Microsoft.
//   All ISVs must use a tag with a 0 in this position.
//   Note: If a Microsoft tag is used by non-Microsoft software, the
//   behavior is not defined.
//
// R is reserved.  Must be zero for non-Microsoft tags.
//
// N is name surrogate. When set to 1, the file represents another named
//   entity in the system.
//
// D is the directory bit. When set to 1, indicates that any directory
//   with this reparse tag can have children. Has no special meaning when used
//   on a non-directory file. Not compatible with the name surrogate bit.
//
// The M and N bits are OR-able.
// The following macros check for the M and N bit values:
//
```

There are reparse point tags for which the **D** bit is not set that may still be used on non-empty directories (they existed prior to this change). Thus, in order to test for this situation a driver should use the new routine (defined in ntifs.h):

```
NTKERNELAPI

BOOLEAN

FsRtlIsNonEmptyDirectoryReparsePointAllowed(

   _In_ ULONG ReparseTag

);
```

In addition, there are new options to control the behavior of opening a directory with such a reparse point (defined in ntifs.h):

```
//  The following flags control behavior when a reparse point is encountered
//  on a directory that may be non-empty (one whose reparse tag is
//  recognized by FsRtlIsNonEmptyDirectoryReparsePointAllowed):
//
//    OPEN_REPARSE_POINT_REPARSE_IF_CHILD_EXISTS -
//    If the reparse point is on a directory that is not the final path
//    component and the next path component exists, reparse on the directory.
//
//    OPEN_REPARSE_POINT_REPARSE_IF_CHILD_NOT_EXISTS -
```

(CONTINUED ON PAGE 14)

File System Related Changes...(Cont.)

(CONTINUED FROM PAGE 13)

```
//    If the reparse point is on a directory that is not the final path
//    component and the next path component does not exist, reparse on the
//    directory.
//
//    OPEN_REPARSE_POINT_REPARSE_IF_DIRECTORY_FINAL_COMPONENT -
//    If the reparse point is on a directory that is the final path
//    component, reparse on the directory unless FILE_OPEN_REPARSE_POINT
//    is specified.
//
//  Specifying all three of the above flags is legal and simply means always
//  reparse on any directory reparse point.
//

#define OPEN_REPARSE_POINT_REPARSE_IF_CHILD_EXISTS              (0x00000002)

#define OPEN_REPARSE_POINT_REPARSE_IF_CHILD_NOT_EXISTS          (0x00000004)

#define OPEN_REPARSE_POINT_REPARSE_IF_DIRECTORY_FINAL_COMPONENT (0x00000008)

#define OPEN_REPARSE_POINT_VERSION_EX                           (0x80000000)
```

This is one of the more interesting and potentially significant change to impact file system mini-filter drivers in RS1 and S16, since it is a change in behavior. Previous releases did not permit attaching reparse points to directories. This release now does. This mechanism can then be used to detect directories where functionality or content is layered, such as in the new container support. Filter drivers that have assumed directories with reparse points are empty must change to accommodate this new model if it impacts their functionality.

Buffer Flushing

NtFlushBuffersFileEx (defined in ntifs.h) now supports a new flush flag (defined in ntddk.h, ntifs.h and wdm.h, and even winnt.h) which is implemented by NTFS:

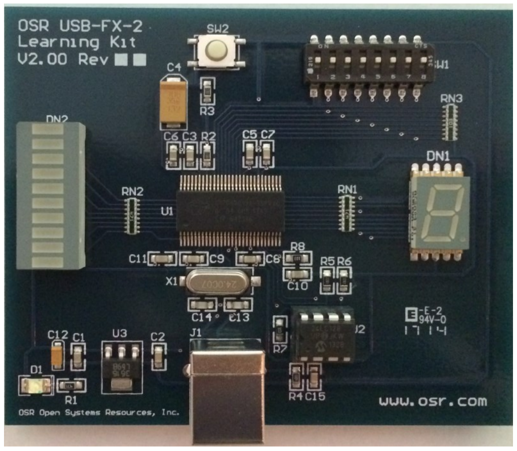
```
//
//  If set, this operation will write the data for the given file from the
//  Windows in-memory cache.  It will also try to skip updating the timestamp
//  as much as possible.  This will send a SYNC to the storage device to flush its
//  cache.  Not supported on volume or directory handles.  Only supported by the NTFS
//  filesystem.
//

#define FLUSH_FLAGS_FILE_DATA_SYNC_ONLY                 0x00000004
```

(CONTINUED ON PAGE 15)

OSR USB FX2 LEARNING KIT

Don't forget, the popular OSR USB FX2 Learning Kit is available in the Store at: http://store.osr.com.

The board design is based on the well-known Cypress Semiconductor USB FX2 chipset and is ideal for learning how to write Windows drivers in general (and USB specifically of course!). Even better, grab the sample WDF driver for this board, available in the Windows Driver Kit.



File System Related Changes...(Cont.)

(CONTINUED FROM PAGE 14)

Thus, this ensures that the data is flushed both from the disk cache and from the CPU cache and persistently stored on disk, ideally using the underlying disk primitives (e.g., force unit access, FUA, when available) to optimally ensure that the blocks for this file are stored persistently on disk.

There is a corresponding bit in the minor function code for IRP_MJ_FLUSH_BUFFERS for this (defined in ntddk.h):

```
#define IRP_MN_FLUSH_DATA_SYNC_ONLY        0x04     //see FLUSH_FLAGS_FILE_DATA_SYNC_ONLY for definition of how
this works
```

There is also a new I/O stack location bit for requesting asynchronous flush behavior (wdm.h):

```
//
//  IRP_MJ_FLUSH_BUFFERS
//

#define SL_FORCE_ASYNCHRONOUS        0x01

//
// SL_FORCE_ASYNCHRONOUS - a flush IRP specific flag in IrpStack to specify that the flush operation needs
// to be async. This behavior is needed by Spaces as Spaces issues flushes to disks in a pool serially and
// does not want to be blocked by disks whose flush operation is slow.
//
```

This impacts both file systems, which may choose to implement this new operation, as well as file system mini-filters, which should ensure they respect the behavior expected by any component using this interface.

This type of interface, permitting applications to control the caching behavior of their files, is important in high reliability systems such as databases where it is important for correctness to ensure that the data has been committed to storage. General purpose applications should not use this because of the potential performance impact.

Correlation IDs

Correlation IDs are GUIDs that are used to uniquely identify a device across the volume stack, permitting event correlation. There is a new routine for obtaining a volume's correlation ID:

```
//
//  Routine to get a correlation ID (currently a GUID) that is common across
//  the volume stack and can be used to correlate events.
//

NTSTATUS
FsRtlVolumeDeviceToCorrelationId (

  _In_ PDEVICE_OBJECT VolumeDeviceObject,

  _Out_ GUID *Guid

  );
```

While there is no documentation about this routine yet, there *is* a code sample in the CDFS source code:

```
        //
        // Initialize the correlation ID.
        //

    if (NT_SUCCESS( FsRtlVolumeDeviceToCorrelationId( Vcb->TargetDeviceObject, &VolumeCorrelationId ) )) {

            //
            // Stash a copy away in the VCB.
            //

      RtlCopyMemory( &Vcb->VolumeCorrelationId, &VolumeCorrelationId, sizeof( GUID ) );
        }
```

(CONTINUED ON PAGE 16)

File System Related Changes...(Cont.)

(CONTINUED FROM PAGE 15)

Its use is for telemetry. There is no matching code in the FastFat example, so it is not clear how widespread its usage is. This is useful for drivers that need to associate specific information with a given volume in a persistent way, even if the file system instance on top of the volume might change, or if a single volume might be presented to the operating system multiple times.

Maximum Path Length Behavior

Note that as of RS1, Windows 10 now has a new registry parameter that lifts the Win32 260-character file name length limitation (**MAX_PATH**). While this is not directly a kernel level change, it does indicate that file system components need to be carefully scrutinized to ensure they can handle long paths.

The new registry key (a DWORD) is:

```
        HKLM\SYSTEM\CurrentControlSet\Control\FileSystem LongPathsEnabled
```

There's also a Group Policy that can be used to lift the 260 character limit. Look under:

```
        Computer Configuration > Administrative Templates > System > Filesystem
```

The value to enable is **Enable Win32 Long Paths**.

The long paths setting is loaded during execution of the first Win32 file system API and is cached by Win32 for the lifetime of the process.

Prior to RS1, you could enable long paths for NTFS for UWP apps and specifically manifested Win32 apps.

How much change this means for file systems and mini-filters is subject to debate. It's always been possible to use paths longer than MAX_PATH, as long as the path specified in UNC syntax (that is, the path started with \\?\). So, while file systems and mini-filters have always technically needed to be able to handle long path names, many probably never saw long paths "in the wild."

File Deletion (Disposition)

Microsoft has introduced three new **FILE_INFORMATION_CLASS** types in Windows (defined in wdm.h):

```
        FileDispositionInformationEx,            // 64
```

This involves introducing a new data structure, which is just a union of flags values (defined in ntddk.h):

```
#define FILE_DISPOSITION_DO_NOT_DELETE           0x00000000
#define FILE_DISPOSITION_DELETE              0x00000001
#define FILE_DISPOSITION_POSIX_SEMANTICS         0x00000002
#define FILE_DISPOSITION_FORCE_IMAGE_SECTION_CHECK  0x00000004
#define FILE_DISPOSITION_ON_CLOSE             0x00000008
typedef struct _FILE_DISPOSITION_INFORMATION_EX {
  ULONG Flags;
} FILE_DISPOSITION_INFORMATION_EX, *PFILE_DISPOSITION_INFORMATION_EX;
```

Some Windows mini-filter samples have been updated to include support for this new type of disposition, including the delete filter and name change filter. Unfortunately, the FastFat sample has not been updated to support this new feature.

This change is introduced to allow managing more complex delete behavior that is apparent with the Linux subsystem on Windows.

(CONTINUED ON PAGE 17)

File System Related Changes...(Cont.)

(CONTINUED FROM PAGE 16)

Specifically, in Linux a file is deleted using **unlink**. Once deleted, any open handles to the file remain valid and continue to work. There is, however, no longer an entry within the directory for that file and thus the name may be reused.

Traditionally in Windows, file deletion is an *intention* that is not acted upon until the last open handle to the file is closed. Indeed, in some scenarios, it is actually possible for an application to undo the intention, in which case the deletion does not occur. Until the last handle is closed, there remains an entry in the directory and the file name cannot be reused.

These semantics do not mesh particularly well with one another. Thus, Windows has changed to provide more nuanced behavior to bridge between them. With this new behavior, the directory entry is deleted as soon as the handle where the file was deleted is closed.

As it turns out, however, due to a compatibility issue this functionality was disabled prior to RS1 release. The Microsoft team have fixed the compatibility issue so it is once again enabled in current test builds of Windows, and is expected to be enabled in the next major Windows 10 update ("Redstone 2" AKA RS2).

Rename

RS1 includes two new rename options (defined in wdm.h):

```
  FileDispositionInformationEx,                 // 65
  FileRenameInformationExBypassAccessCheck, // 66
```

The new information class **FileRenameInformationExBypassAccessCheck** is comparable to **FileRenameInformationBypassAccessCheck**. This is consumed by the I/O Manager and has the effect of disabling the security check associated with the rename operation, which can cause a deletion of the target file. Note that there is no new data structure, as it utilizes previously unused pad space within the rename structure (defined in ntifs.h):

```
typedef struct _FILE_RENAME_INFORMATION {
#if (_WIN32_WINNT >= _WIN32_WINNT_WIN10_RS1)
    union {
    BOOLEAN ReplaceIfExists;  // FileRenameInformation
        ULONG Flags;         // FileRenameInformationEx
  } DUMMYUNIONNAME;
#else

  BOOLEAN ReplaceIfExists;
#endif

  HANDLE RootDirectory;
  ULONG FileNameLength;
  WCHAR FileName[1];
} FILE_RENAME_INFORMATION, *PFILE_RENAME_INFORMATION;
```

And the corresponding new flags (defined in ntifs.h):

```
#define FILE_RENAME_REPLACE_IF_EXISTS                0x00000001
#define FILE_RENAME_POSIX_SEMANTICS                  0x00000002
```

This preserves the previous semantics and adds the new POSIX semantics. Much like the changes in delete, these are introduced to deal with the variation in behavior between Linux and Windows subsystems. In Windows, a destructive rename will fail if the

(CONTINUED ON PAGE 18)

File System Related Changes...(Cont.)

(CONTINUED FROM PAGE 17)

file is currently opened. In Linux, it will succeed. The open handle is still valid and continues to work, even though there is no longer an entry in the directory pointing to it.

For POSIX semantics to work, the application that has it open must have specified **FILE_SHARE_DELETE** when the file was opened. Otherwise the rename will fail.

Microsoft has updated some of the mini-filter examples to demonstrate how to handle the new rename type, including the context filter, and name changer filter. Fortunately, the impact for mini-filters is likely to be minimal, unless your filter needs to understand the new semantic behavior differences. In such a case, you would need to adjust your mini-filter accordingly.

Filter Manager

There were a number of small changes in Filter Manager in Redstone. These changes were all in fltKernel.h.

The first involves the **FLT_VOLUME_PROPERTIES** where a previously reserved field has been converted to a flags field:

```
  USHORT Flags;
```

One flag is currently defined:

```
//
//  FLT_VOLUME_PROPERTIES Flags
//
//  VOL_PROP_FL_DAX_VOLUME - If set, this is a DAX volume i.e. the volume supports
//  mapping a file directly on the persistent memory device.  The cached and memory
//  mapped IO to user files wouldn't generate paging IO.
//

#define VOL_PROP_FL_DAX_VOLUME              0x0001
```

A new operation for obtaining the attribution handle from the callback data was introduced:

```
_IRQL_requires_max_(DISPATCH_LEVEL)

PVOID
FLTAPI
FltGetIoAttributionHandleFromCallbackData (

    _In_ PFLT_CALLBACK_DATA Data

    );
```

And a mechanism for “propagating” IRP extension data between two callback data structures:

```
_IRQL_requires_max_(DISPATCH_LEVEL)
NTSTATUS
FLTAPI
FltPropagateIrpExtension (
  _In_ PFLT_CALLBACK_DATA SourceData,
  _Inout_ PFLT_CALLBACK_DATA TargetData,
  _In_ ULONG Flags
    );
```

Note that IRP extensions were first added in Windows 10 (1511). Neither of these two calls are documented yet. Attribution was added for Windows 10 (1607) and is used as part of I/O rate management for containers.

As previously mentioned, Filter Manager now has a new flag that a mini-filter uses to indicate that it wishes to be notified about direct access storage volumes:

FLTFL_REGISTRATION_SUPPORT_DAX_VOLUME

Note that a filter which does not set this flag will not be asked to attach to such volumes.

(CONTINUED ON PAGE 19)

File System Related Changes...(Cont.)

(CONTINUED FROM PAGE 18)

Filters that perform secondary operations will need to keep this new mechanism in mind so that the attribution handle can be properly reflected between otherwise distinct calls. A failure to do this will interfere with the I/O Rate Control Driver (iorate.sys) , which uses this information. Thus, if your filter driver will be running on Windows Server 2016 systems, it is important to ensure you are properly passing along this information.

Summary

The Windows RS1 and S16 releases introduce a number of interesting new changes and while some are clearly described and documented, some also remain unclear to us at the present time. Rest assured that as we expand our understanding of them, we will be sure to let you know as well!

Special thanks to Microsoft's Shoily Rahman for assistance in completing our understanding of the Delete/Rename changes.

Follow us!

WINDOWS FILE ENCRYPTION? WE CAN HELP!

OSR'S FILE ENCRYPTION SOLUTION FRAMEWORK (FESF)

We know that implementing per-file encryption solutions for Windows is challenging—and we've been doing it for more than 20 years!

Of course, it's also a challenge to convince devs and their managers of the difficulties they face. Many are new to the Windows kernel development space. They see sample mini-filters in the WDK or may have even written one themselves. It's an easy jump (in their minds) to, "Hey...how hard can it be to deal with encrypting the data I'm filtering?" Even worse, with some help in NTFSD, they may spend only a few months getting a solution up and running - solving "one more bug" or dealing with "one more crash" - without realizing that they are just beginning to scratch the surface of the difficulties in getting from a prototype to a real file encryption product.

Please don't let this be you. OSR can help save you a great deal of time and money and help make your Windows file encryption solution successful. OSR's latest toolkit—the File Encryption Solution Framework (FESF) builds on a time-tested infrastructure, but moves all the core development for a Windows file encryption solution to USER mode. You don't need to be an expert in Windows file system or kernel programming, and your time is better spent defining "policy" of your solution (what you want to encrypt, when, and with what algorithm and key management) instead of wrestling with the subtle and painful nuances in filtering file systems.

TRY A FULLY FUNCTIONAL EVALUATION EDITION OF FESF FOR FREE!

Want to try a fully-functional evaluation of FESF for FREE? Just contact the OSR sales team and they'll get you started.

Contact: sales@osr.com

Time Saving Strategies... (Cont.)

(CONTINUED FROM PAGE 9)

Building a sustainable development and testing infrastructure in user mode tends to be *much* more work. But it can also yield *much* bigger returns on your invested time and effort. As needs change or bugs are found, you can return to your quick and convenient user mode infrastructure to rapidly code and/or test those changes.

The biggest problem that we've seen in maintaining a sustainable user mode infrastructure is that it's very easy for code paths to diverge and "rot." That is, once the initial user mode portion of your project has been completed and you've moved on to kernel mode development and testing, you'll certainly be making changes to your driver's code modules. It can be pretty easy for these changes to unintentionally break – in small or large ways – the user mode infrastructure that you previously created. If you don't build and exercise the user mode infrastructure regularly, resurrecting it after a significant series of kernel mode changes can be quite a challenge. Worse yet, while you're in the midst of your development cycle driving to a release, maintaining the user mode infrastructure can feel like an unnecessarily burdensome task.

For the projects that I've done, I almost always favor the "use once and throw away" approach. I find if I *don't* take that approach, I spend way too much time thinking about my user mode infrastructure. If it's going to be a persistent part of the project, I feel like I have to "design" it; I feel like I have to do a professional job of it. If it's something that's going to be used once and thrown away, I feel more at ease with just "making it work." Hand me that chainsaw, please.

```
#ifndef _KERNEL_MODE

#pragma once

#ifdef _DEBUG

#define DBG 1

#endif

#ifdef __cplusplus
extern "C" {
#endif

#define CLONG ULONG

#ifndef _AMD64_    // UM supports x64 only
#define _AMD64_ 1
#endif

#define UMDF_USING_NTSTATUS 1
#include <stdio.h>
#include <Windows.h>
#include <ntstatus.h>

#define DbgPrint printf

//
// Interrupt Request Level (IRQL)
//

typedef UCHAR KIRQL;

typedef KIRQL *PKIRQL;

//
// AMD64 Specific portions of Mm component.
//
// Define the page size for the AMD64 as 4096 (0x1000).
//

#define PAGE_SIZE 0x1000

//
// Define the number of trailing zeroes in a page aligned
// virtual address. This is used as the shift count when
// shifting virtual addresses to virtual page numbers.
//

#define PAGE_SHIFT 12L

#define NT_SUCCESS(Status) (((NTSTATUS)(Status)) >= 0)
```

... File continues...

Figure 1—Header File for User Mode Code

First Practical Steps

Whether you decide to create a one-time or a lasting infrastructure, there are some simple, practical, hints that we can provide that'll make your project easier.

First, unless processor architecture (x64, x86, or ARM) plays a crucial role in the algorithms you'll be developing and testing in user mode, decide on one processor architecture for your user-mode infrastructure and just stick to that. You ever look at the code in NTDDK.H that's conditionalized based on processor architecture? Yeah… you don't want to try to recreate that and keep it maintained unless absolutely necessary.

Next, we recommend that you create a dedicated header file that will contain most of the definitions for your project that are specific to user mode. Conditionalize this header on the code not being built in kernel mode. Our user-mode specific headers usually start somewhat as shown in *Figure 1*.

Looking at Figure 1, you'll notice that we qualify the entire include file by the symbol **_KERNEL_MODE** *not* being defined. To get the NTSTATUS values defined, we include NTSTATUS.H, but only after defining UMDF_USING_NTSTATUS to 1. We don't recommend you try to #include any of the other WDK header files. What we've found works best is including WINDOWS.H and then just copying the definitions you need from NTDDK.H or

(CONTINUED ON PAGE 21)

Time Saving Strategies... (Cont.)

(CONTINUED FROM PAGE 20)

WDM.H into your dedicated user mode header file. Sure it's ugly, but you won't lose any points for including stuff your code doesn't really need. So, copy and paste some stuff when you start, and then as you find you need things defined – be they function prototypes, macros, or typedefs – just add them to your user mode header file and you're good to go.

There's one set of macros that's used by almost every driver that's been written: The list manipulation macros. These include **InsertTailList**, **RemoveHeadList**, and all their friends. These are defined in WDM.H. A quick hint is that it'll be simpler to copy the definitions if you copy the versions from WDM.H that are defined when the **NO_KERNEL_LIST_ENTRY _CHECKS** is defined. This leaves out the dynamic checks for list consistency, which you probably won't need.

Once you get the basics defined, I think you'll be surprised at how little you need to cut/paste from the WDK headers. If your code is focused on algorithms and data structure manipulation, most of that type of code doesn't tend to use a lot of kernel mode specific functions.

Allocating Memory, Locks, and Stuff

The most common question that we encounter in building a user-mode driver testing infrastructure is how to handle basic functions that are different in kernel mode and in user mode. For example, when the code that you're writing, and that you'll eventually be moving to kernel mode, needs to allocate memory or perform locking functions. Calls to these functions tend to be spread all through your driver code. How do you handle the fact that the names of the routines that you need to call in user mode are different from the names in kernel mode?

There are two possible approaches, and both have been used by teams here at OSR. The easiest approach is to use macros to define the kernel mode function name as some reasonable user mode equivalent. So, in terms of allocating memory, you might put the following definition in your dedicated user mode header file:

```
#define ExAllocatePoolWithTag(size, tag)    malloc(size)
```

You then write your code to use the kernel mode function name, and your dedicated user mode header "does the right thing" by defining it for user mode use.

(CONTINUED ON PAGE 22)

Time Saving Strategies... (Cont.)

(CONTINUED FROM PAGE 21)

Alternatively, you might choose to create private functions that perform the necessary operation(s), and "do the right thing" within those functions based on the mode for which the code is being compiled. For memory allocation, you might define a function like the following (see *Figure 2)*.

```
__forceinline
PVOID
CachePackageAllocateMemory(_In_ SIZE_T NumberofBytes, _In_ ULONG Tag)
{
#ifdef _KERNEL_MODE

    return(ExAllocatePoolWithTag(NonPagedPoolNx, NumberofBytes, Tag));

#else

    UNREFERENCED_PARAMETER(Tag);

    return(VirtualAlloc(NULL, NumberofBytes, MEM_COMMIT, PAGE_READWRITE));

#endif
}
```

Figure 2—Use of a Private Function (e.g., for memory allocation)

Personally, I tend to favor the pattern of defining private functions for things like memory allocation even when I'm developing and testing my code strictly in kernel mode. There's something about isolating the underlying mechanism from the need for that mechanism that just appeals to me. Of course, if you just want to get something running so you can test in user mode, it's hard to argue with the simplicity of just writing your code to reference the familiar kernel mode function names you already know and love, and then redefining those names as their chosen user mode equivalents. The choice is really up to you, and is mostly related to the coding practice you prefer.

The same alternatives apply to locking. You could choose to simply #define the kernel mode function names as some appropriate user mode equivalent. But, once again, I often choose to define private locking functions in my driver code that's particular to specific structures. Whichever method you choose, it's usually pretty simple to find user mode locks that have similar semantics to the type of lock you choose in kernel mode. In *Figure 3*, you can see how we've handled a shared reader/writer lock that would be usable at elevated IRQL in kernel mode.

(CONTINUED ON PAGE 23)

```
_Requires_lock_not_held_(GCLock->Lock)
_Acquires_shared_lock_(GCLock->Lock)
__forceinline
VOID
OSRCachePackageAcquireLockShared(_In_ PGC_LOCK GCLock, _Out_ PKIRQL OldIrql)
{
#ifdef _KERNEL_MODE

    *OldIrql = ExAcquireSpinLockShared(Lock);

#else
    AcquireSRWLockShared(&GCLock->Lock);
    *OldIrql = 0;
#endif
}

_Requires_shared_lock_held_(GCLock->Lock)
_Releases_shared_lock_(GCLock->Lock)
__forceinline
VOID
OSRCachePackageReleaseLockShared(_In_ PGC_LOCK GCLock, _In_ KIRQL OldIrql)
{
#ifdef _KERNEL_MODE

    ExReleaseSpinLockShared(Lock, OldIrql);

#else
    UNREFERENCED_PARAMETER(OldIrql);
    ReleaseSRWLockShared(&GCLock->Lock);
#endif
}
```

Figure 3—Use of Equivalent Locking in User Mode

Time Saving Strategies... (Cont.)

(CONTINUED FROM PAGE 22)

Figure 3 only shows one pair of lock/unlock functions, but that should be enough to illustrate the overall idea without making things too boring. Note that we define our own, private, data type for the lock (GC_LOCK in the example). The definition of this data type will vary, based on whether the code is compiled for user mode or kernel mode. This allows us to allocate storage for the lock in our data structures without having to hand-code changes in those structures. In the example, you can see that we use reader/writer spin locks in kernel mode, and in user mode we've chosen Slim Reader/Writer locks as a reasonable analogue. Note again that, we conditionalize the code in the private function (which typically lives in a component-specific header file, along with the mode-specific definition of the lock data type itself).

Whatever type of kernel mode lock you need, you'll be able to easily find a user mode analogue. And, again, whether you choose to acquire and release those locks using appropriately conditionalized private functions that you define or just #define the kernel function names to their user mode equivalents, is probably more to do with your engineering style than whether one method can be considered objectively superior.

Support Routines

One major advantage of building your test infrastructure in user mode is that you have access to all the "stuff" that coding in user-mode provides. Need random numbers, some type of collection to store test data, or the ability to sort your results? You have everything that the C++ Standard Library (std::) has to offer available to you. Need to allocate enormous data structures? Just malloc/new to your heart's content. Need to read or record test data? Easy access to file I/O is a good thing. Need six million local variables, including an array with a zillion entries? Just declare it all and let the user mode stack grow to accommodate it. I'm not saying it's *impossible* to handle these things in kernel mode. I'm just saying things such as these are a *lot* easier in user mode.

But wait, there's more! Not only do you have the benefit of access to all the support routines that user mode has to offer, you can still use your familiar friends from kernel mode, the Run Time Library (RTL) functions. Most of the RTL functions are available for use in your user mode test harness by linking with NTDLL.LIB. So, for example, if you decide to use the RTL's Generic Table Package there's no problem with calling those functions. Love the RTL-defined bit map routines? They're there for you to call. Of course, you *will* have to copy the prototypes for these functions from the appropriate WDK header file to your dedicated user mode definition file.

Other Miscellaneous Advantages

I hate to say it, but programming and debugging in user mode using Visual Studio brings with it a lot of life-simplifying features. The user-mode debugger knows a few tricks that the kernel mode debugger hasn't learned yet. When running in user mode, you can also use the lovely little "performance profiler" that's integrated into Visual Studio to evaluate both CPU and memory usage. I actually found a bug using the memory profiler recently, so it has demonstrated its worth to me (at least once).

It's a Win!

For the right driver project, developing and testing algorithm and data structure intensive routines in user mode can save time while increasing the thoroughness of your testing. By writing a bit of "bridge logic" to ensure your favorite functions and data structure are available in user mode, you can avail yourself of all the features that coding in user mode provides: Richer support routines, a slightly nicer debugging experience, and – perhaps most importantly of all – a much faster time for each iteration through the "build/test/find-bug/fix-bug" cycle.

Who would think that one of the newest things in the world of kernel mode driver development is... user mode development and testing!

Making Device Objects Accessible... (Cont.)

(CONTINUED FROM PAGE 7)

Device Setup Class

In addition to device name, a driver can also specify the Device Setup Class that is associated with the Device Object that is being created. The Device Setup Class is, among other things, the category under which the device is shown in Windows Device Manager. Note that the Device Setup Class is also specified in the INF file that is used to install the driver. Specifying a Device Setup Class from your driver allows Windows to locate class-specific default settings for the Device Object that is being created. Such settings can include device type, device characteristics, and most importantly the security descriptor to be applied to the newly created Device Object within the class.

Specifying the Device Setup Class when you create an FDO is both simple and a best practice. As in specifying an internal name, the method uses the **WDFDEVICE_INIT** structure, and can be invoked as follows:

```
WdfDeviceInitSetDeviceClass(DeviceInit,
                            &GUID_DEVCLASS_OSR_MISSILE);
```

In this example, the caller calls **WdfDeviceInitSetDeviceClass** to set the Device Setup Class to GUID_DEVCLASS_OSR_MISSILE. Note that because this is a properties method (the verb used is “set”) it cannot fail. Thus, there is no status returned from the call.

Specifying the Device Setup Class in your driver gives you the best chance of Windows taking the device type, device characteristics, exclusive access, and – most importantly – the security descriptor that is applied to your FDO and “harmonizing” them throughout your device stack. By “harmonize” we mean that it takes the settings from your FDO and copies them to the PDO (and, presumably, any other Device Objects in your device stack). We’ll explain more about this later.

Device Object Security

The final characteristic of interest that can be specified when creating a Device Object is the security descriptor. This allows you to define the access rights that specific users and groups will have to the Device Object you’re creating. Again, this operation is performed using a method on the WDFDEVICE_INIT structure, like the following:

```
status = WdfDeviceInitAssignSDDLString(DeviceInit,
                                        &SDDL_DEVOBJ_SYS_ALL_ADM_ALL);
```

The method is **WdfDeviceInitAssignSDDLString**. Note that the verb used in this method is “assign” so the call *can* fail, a status value *is* returned, and that status *must* be checked. The security descriptor is specified using a limited form of Security Descriptor Definition Language (SDDL). SDDL is a short-hand method of specifying standard Windows access control specifications. The sub-set of SDDL that is used for Device Objects allows most of the common options, but also provides a few pre-defined values for commonly used security profiles. For example, the option shown in the example is SDDL_DEVOBJ_SYS_ALL_ADM_ALL. This provides System and Administrators all access to the device, but no access to any other class of user. There are numerous other shorthand specifications that allow a wide variety of access combinations.

While specifying an SDDL string to apply a specific security descriptor to your Device Object probably sounds like a good way to ensure security, it turns out that *specifying this option is almost never a good idea*. In fact, best practices call for *not* specifying an SDDL string using **WdfDeviceInitAssignSDDLString** (we’ll explain why a bit later). If you want to change the default protection applied to the Device Object’s in your device stack, the best way to do that is to specify a security descriptor during device/driver installation. You do this in your INF either as part of the DDInstall.HW section (where the security descriptor is applied to your specific device) or in the ClassInstall32 section (where the security descriptor is applied to all the devices in your Device Setup Class). In either case, when you specify a security descriptor in your INF file, it is stored in the Registry and later used for your device, as appropriate. See the sidebar **Specifying Security in the INF File**.

There are two important, and probably unexpected, things to keep in mind about using **WdfDeviceInitAssignSDDLString**:

- To create a Device Object successfully after having called **WdfDeviceInitAssignSDDLString**, you must also specify an internal name for your device. So, if you call **WdfDeviceInitAssignSDDLString** you *must* also call **WdfDeviceInitAssignName** (or request that Windows autogenerate a device name for the Device Object you’re creating).

(CONTINUED ON PAGE 25)

Making Device Objects Accessible... (Cont.)

(CONTINUED FROM PAGE 24)

- The protection you specify when you call **WdfDeviceInitAssignSDDLString** is the *last-chance default* protection. *It will only be used if a security descriptor for your device or Device Setup Class has not been previously stored in the Registry*. In the case that there is no security descriptor already stored in the Registry, the security descriptor you specify will be stored in the registry and become the new default security descriptor for Device Objects subsequently created in the Device Setup Class (assuming you have specified a Device Setup Class using **WdfDeviceInitSetDeviceClass)**.

The reason it is best practice to *not* call **WdfDeviceInitAssignSDDLString** is because the specified security descriptor is only used when there is no default security descriptor setting already known for the device or the Device Setup Class. This is true, even if the security descriptor specified by the driver is less permissive than the one that has been previously stored in the Registry. So, even when you specify a security descriptor in your driver, there's no guarantee that it will define the security that is actually used on your Device Object. And if that is the case, why specify it at all?

If you do not specify a security descriptor for your Device Object by calling **WdfDeviceInitAssignSDDLString**, and if there's no default stored in the Registry for your device or your Device Setup Class, the system will assign a security descriptor. For FDOs, the Framework assigns SDDL_DEVOBJ_SYS_ALL_ADM_ALL, which provides access only to system and administrators as described above. For PDOs, Windows provides the default based on Device Type. In general, this protection allows all users read and write access to the device.

Making Devices Accessible

After characteristics and properties are established and a Device Object is created, how do user-mode applications access the device?
A user-mode application opens and sends I/O operations to a Device Stack by opening a Device Object in the stack that is been explicitly made accessible to user-mode by a kernel-mode module. Drivers can make either the FDO or the PDO accessible, or both. There are three different ways that drivers can make Device Objects accessible to user-mode applications:

- Create a symbolic link to the PDO in the Device Stack
- Create a symbolic link to the FDO in the Device Stack
- Create a Device Interface GUID that points to the PDO in the Device Stack

All of these mechanisms result in the user-mode application eventually calling the Win32 function **CreateFile** to access a Device Object.

To make either the PDO or FDO easily accessible by name to user-mode applications, the driver explicitly creates a symbolic link for the device by calling WdfDeviceCreateSymbolicLink:

```
DECLARE_CONST_UNICODE_STRING(userDeviceName, L"\\DosDevices\\ML1");
status = WdfDeviceCreateSymbolicLink(device, &userDeviceName);
```

(CONTINUED ON PAGE 26)

Making Device Objects Accessible... (Cont.)

(CONTINUED FROM PAGE 25)

The method **WdfDeviceCreateSymbolicLink** is, obviously, a method on a WDFDEVICE, and thus must be called after the Device Object has been created. In the example above, the driver creates a symbolic link in the Object Manager's \DosDevices\ namespace, which will allow a user-mode application to access the device using code similar to the following:

```
hFile = CreateFile("\\\\.\\ML1",
                   GENERIC_READ|GENERIC_WRITE,    // requested access
                   0,                             // share mode
                   NULL,                          // security attributes
                   OPEN_EXISTING,                 // create disposition
                   0,                             // flags
                   NULL);                         // template file
```

Note all the ugly doubled backslash characters, as required by C syntax. The "\\.\" syntax represents a UNC path and is required. Note that it *is* possible for user-mode applications to access Device Objects even without such a symbolic link being created. But it's ugly and/or arbitrary. In any case, it's certainly not convenient for the app developer, and convenience is the primary reason for creating the symbolic link in the first place.

The tricky thing for driver devs about **WdfDeviceCreateSymbolicLink** is that what it does varies depending on whether you've named your FDO or not. Yes, seriously.

If you have not named your FDO, that is you did not call **WdfDeviceInitAssignName** prior to creating your Device Object, *the symbolic link that is created will point to the PDO*.

If your driver chose to provide an internal name for your FDO by calling **WdfDeviceInitAssignName** prior to creating your Device Object, *the symbolic link that is created will point to the FDO*.

The third way a driver can make a device accessible is through the use of a Device Interface GUID. After the Device Object has been created, the function driver tells the Framework that the Device Object that was created supports a given Device Interface GUID. It does this via the function **WdfDeviceCreateDeviceInterface**. Supporting a given Device Interface implies the services provided by the driver on behalf of the device and even the I/O function codes (including IOCTLs) the device supports. There are numerous system-defined Device Interface Classes (search C:\Program Files (x86)\Windows Kits\10\include\ to get an idea of just how many there are!). So, for example, if a device supports GUID_DEVINTERFACE_COMPORT, you know that it is associated with a traditional PC serial port, and will support all the IOCTL_SERIAL_Xxxx requests.

For custom drivers, the GUID that represents the Device Interface will be one that is been uniquely assigned to a given class of devices. So, in our example of the OSR Missile Launcher device, I would probably define a new Device Interface that exclusively represents that type of device. I'd do this by defining something like GUID_DEVINTERFACE_OSR_MISSILE in a header file that can be shared between my driver and any applications that need to use my device.

Device Interfaces are cool, because they make a number of interesting things possible, including:

- Providing a mechanism for a driver to categorize a device as supporting a particular interface and thus providing a particular type of service. User programs can enumerate which devices in the system support a particular interface without regard to device name.

- Allowing both kernel-mode and user-mode entities to register callbacks that are invoked whenever a Device Interface changes state -- Such as when a new instance of an interface is enabled or an existing interface is disabled.

Another nice thing about Device Interfaces is that when they are used in place of device names, any potential for naming conflicts is avoided. Not that I've ever really had a problem with Device Object naming conflicts. But, whatever.

One of the primary things that driver devs need to keep in mind about Device Interfaces is that *Device Interfaces always point to a PDO*.

(CONTINUED ON PAGE 27)

Making Device Objects Accessible... (Cont.)

```
HANDLE
OpenMissileDeviceViaInterface(VOID)
{
    HDEVINFO                         devInfo;
    SP_DEVICE_INTERFACE_DATA         devInterfaceData;
    PSP_DEVICE_INTERFACE_DETAIL_DATA devInterfaceDetailData = NULL;
    ULONG                            devIndex;
    ULONG                            requiredSize;
    ULONG                            code;
    HANDLE                           handle;

    devInfo = SetupDiGetClassDevs(&GUID_DEVINTERFACE_OSR_MISSILE,
                                  NULL,
                                  NULL,
                                  DIGCF_PRESENT | DIGCF_DEVICEINTERFACE);

    if (devInfo == INVALID_HANDLE_VALUE) {

        printf("SetupDiGetClassDevs failed with error 0x%x\n", GetLastError());

        return INVALID_HANDLE_VALUE;
    }

    devInterfaceData.cbSize = sizeof(SP_DEVICE_INTERFACE_DATA);

    devIndex = 0;

    if (!SetupDiEnumDeviceInterfaces(devInfo,
                                     NULL,
                                     &GUID_DEVINTERFACE_OSR_MISSILE,
                                     devIndex++,
                                     &devInterfaceData)) {

        code = GetLastError();

        if (code != ERROR_INSUFFICIENT_BUFFER) {

            printf("SetupDiGetDeviceInterfaceDetail failed with error 0x%x\n", code);

            SetupDiDestroyDeviceInfoList(devInfo);

            return INVALID_HANDLE_VALUE;
        }
    }

    if (!SetupDiGetDeviceInterfaceDetail(devInfo,
                                         &devInterfaceData,
                                         NULL,
                                         0,
                                         &requiredSize,
                                         NULL)) {

        code = GetLastError();

        if (code != ERROR_INSUFFICIENT_BUFFER) {

            printf("SetupDiGetDeviceInterfaceDetail failed with error 0x%x\n", code);

            SetupDiDestroyDeviceInfoList(devInfo);

            return INVALID_HANDLE_VALUE;
        }
     }

    devInterfaceDetailData =
        (PSP_DEVICE_INTERFACE_DETAIL_DATA)malloc(requiredSize);
```

Figure 2—Opening a Device Interface (continued next page)

(CONTINUED FROM PAGE 26)

When a driver creates a Device Interface, a user-mode application accesses the device by first finding the set of devices that support a given Device Interface GUID. This is done using interfaces provided by the **SetupDiXxxx** family of functions. Once a particular device supporting a given Device Interface is chosen, the user-mode app opens that device using **CreateFile** with an opaque name that the application retrieves from the details of Device Interface instance. Basic code to open the first device found that is associated with a given Device interface is shown in *Figure 2*.

(CONTINUED ON PAGE 28)

Making Device Objects Accessible... (Cont.)

```
    if (!devInterfaceDetailData) {

        printf("Unable to allocate resources...Exiting\n");

        SetupDiDestroyDeviceInfoList(devInfo);

        return INVALID_HANDLE_VALUE;
    }

    devInterfaceDetailData->cbSize =
        sizeof(SP_DEVICE_INTERFACE_DETAIL_DATA);


    if (!SetupDiGetDeviceInterfaceDetail(devInfo,
                                         &devInterfaceData,
                                         devInterfaceDetailData,
                                         requiredSize,
                                         &requiredSize,
                                         NULL)) {

        printf("SetupDiGetDeviceInterfaceDetail failed with error 0x%x\n",
                                             GetLastError());

        SetupDiDestroyDeviceInfoList(devInfo);

        free(devInterfaceDetailData);

        return INVALID_HANDLE_VALUE;
    }

    printf("Device found! %ls\n", devInterfaceDetailData->DevicePath);

    SetupDiDestroyDeviceInfoList(devInfo);

    if (devInterfaceDetailData == NULL) {

        printf("Unable to find any matching devices!\n");

        return INVALID_HANDLE_VALUE;
    }

    handle = CreateFile(devInterfaceDetailData->DevicePath,
                        GENERIC_READ | GENERIC_WRITE,
                        0,
                        0,
                        OPEN_EXISTING,
                        0,
                        0);

    free(devInterfaceDetailData);

    return handle;
}
```

Figure 2—Opening a Device Interface

(CONTINUED FROM PAGE 27)

Kernel-mode modules can also access your Device Object in various ways using the Device Interface GUID you've established. That means if you follow best practices and do not name your FDO, there are still ways other kernel-mode modules can find and access your device.

Security on Open – But on WHICH Device Object?

You'll note that we've been very specific in this article about which Device Object (the PDO or the FDO) a given access option points to. The reason we've done this is because it matters. Not understanding which Device Object a given access point is referring to is the basis for most problems in terms of Device Object protection.

When an application calls **CreateFile**, *the only security descriptor that is checked to determine if the **CreateFile** will succeed is the security descriptor on the Device Object that is being directly opened*. This means, for example, that if the user attempts to access a Device Object via a link that points to the FDO, the access check that will take place will be based on the user's security credentials and the security descriptor on the FDO. Similarly, if the user calls **CreateFile** to access a device via its PDO, the user's credentials are checked against the security descriptor on the PDO.

(CONTINUED ON PAGE 29)

Making Device Objects Accessible... (Cont.)

(CONTINUED FROM PAGE 28)

And remember earlier we said that regardless of whether I/O is sent to the PDO or to the FDO, the I/O operations always are routed first to the top of the Device Stack. *So, as far as the user-mode application's ability to send I/O requests to a driver is concerned, it doesn't matter whether it opens the PDO or the FDO*. All I/O operations will go to the FDO (all other things being equal).

Where Are We?

Let's see if we can summarize some of the rules and best practices we've discussed so far. Let's assume you're writing a function driver. As part of creating your FDO:

- We suggest, as a best practice, you specify a Device Setup Class for your FDO by calling **WdfDeviceInitSetDeviceClass**. This isn't a big deal, really, but it does help to ensure the Device Object protections are "harmonized" across all the Device Objects in your device stack in certain instances. There are no downsides to specifying it, and if you just always do it, you'll find a few edge cases dealing with protection in your device stack are properly handled.

- You may optionally specify an internal name for your FDO, however doing so it not recommended unless absolutely necessary (and is not a best practice).

- If you *do* choose to name your FDO, you can optionally specify a last-chance default security descriptor for the FDO… but, again, it is best practice to *not* do this.

- If you choose to *not* specify a last-chance default security descriptor for your FDO, the protection applied to your FDO will be depend:

  - If there is a default security descriptor stored in the Registry for your device or your Device Setup Class, then that security descriptor will be applied to all Device Objects in your device stack (including the FDO you are creating).

  - If there is no default security descriptor stored in the Registry for your device's Device Setup Class, then:

    - If you have *not* specified a Device Setup Class, then your FDO will receive the default protection provided by the Framework (all access for system and administrators but no access for any other groups) and your PDO will receive the default protection provided by Windows (system and admin all access, everyone else read and write access).

    - If you *have* specified a Device Setup Class (by calling **WdfDeviceInitSetDeviceClass**) the default protection provided by the Framework will be applied *to your FDO and the other Device Objects in your device stack* (including the PDO).

Once your FDO has been created, you can make it accessible to user-mode applications by:

- Creating a symbolic link in the Object Manager's \DosDevices\ name space. User mode applications can open the device via the symbolic link name you provide. If your FDO is named, the symbolic link will point to the FDO. If you did not name your FDO, the symbolic link will point to the PDO.

- Associating your FDO with a Device Interface. This approach has a number of advantages, but requires user-mode applications to use the SetupDiXxx functions with your Device Interface GUID to retrieve the opaque name associated with a specific instance of the Device Interface to open your device. The opaque name used to open a device by Device Interface GUID always points to the PDO.

(CONTINUED ON PAGE 30)

Making Device Objects Accessible... (Cont.)

(CONTINUED FROM PAGE 29)

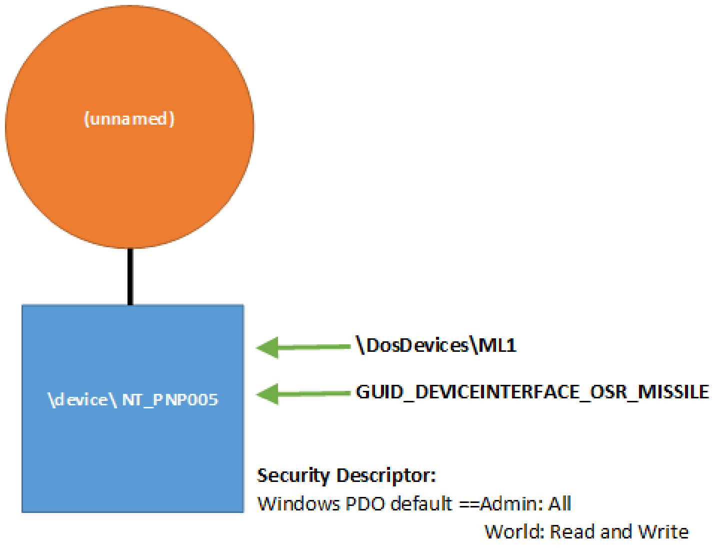


Figure 3—Unnamed FDO, Device Interface Specified, Default Protections

You can make your device accessible using either of the above two methods, or both of them. Here at OSR, we typically use both methods (as do many Windows in-box drivers).

Some Examples

Let's look at a couple of examples as a way to more fully understand the options available to us, and to help us understand the impact of those options.

First, let's say we do not provide an internal name our FDO. We create a symbolic link to allow applications to open our device by name, and we also associate our FDO with a unique Device Interface GUID. We do not specify a last-chance security descriptor for our FDO, nor do we specify a Device Setup Class. The security descriptors on both our PDO and FDO will default to whatever is provided by the system. In this case, we wind up with a device stack like that shown in *Figure 3*.

Looking at Figure 3, you can see that both our Device Interface and our symbolic link name point to the PDO. The symbolic link name we provide by calling **WdfDeviceCreateSymbolicLink** points to the PDO because we did not specify an internal name for our FDO. Recall that regardless of whether the names point to the FDO or PDO, all I/O operations from the application will go to the top of the device stack. So all I/O requests (such as read, write, and DeviceIoControl) sent by the application will always to go the FDO first.

Now, let's see what happens when we choose to provide an internal name for our FDO, by calling **WdfDeviceInitAssignName**. This name will allow other kernel-mode modules to access our device. In this case, we'll do everything the same as we did in Figure 3: We create a symbolic link to allow our device to be opened by name from user-mode applications, and we associate our device with a Device Interface GUID. And, once again, we do not specify a last-chance security descriptor and we do not specify a Device Setup Class. Again, the security descriptors on both our PDO and FDO will default to those provided by the system. In this case, we wind up with a device stack like that shown in *Figure 4*.

There are several important changes from Figure 3 to notice in Figure 4. First, notice that the symbolic link name (ML1) now points to the FDO instead of to the PDO. This is because we chose to provide an internal name for our FDO. The Device Interface still points to the PDO (as it always will).

Next, notice the security descriptors for the FDO and PDO are different. If we name our FDO, and there's no Device Setup Class security descriptor available, the Framework provides an "administrator only" security descriptor for our FDO by default.

As a result of the above two decisions, which security descriptor applies when an application issues a **CreateFile** will depend on whether the application attempts to open our device by symbolic link name or by the opaque name obtained via the Device Interface GUID. To be able to successfully open our

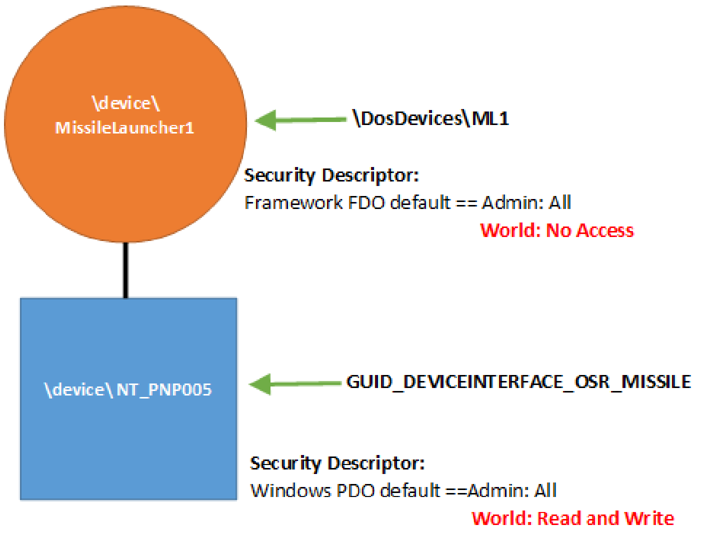

\device\ MissileLauncher1
\DosDevices\ML1
Security Descriptor:
Framework FDO default == Admin: All
World: No Access
\device\ NT_PNP005
GUID_DEVICEINTERFACE_OSR_MISSILE
Security Descriptor:
Windows PDO default ==Admin: All
World: Read and Write

Figure 4—*Named* FDO, Device Interface Specified, Default Protections

(CONTINUED ON PAGE 31)

Making Device Objects Accessible... (Cont.)

(CONTINUED FROM PAGE 30)

device by symbolic link name, an application will require administrator privileges (that is, the app will need to be run elevated, “as administrator”). This is because the symbolic link name points to the FDO, and the security descriptor on the FDO requires administrator access.

However, that same application would not need administrator privileges to be able to open the device using the Device Interface. Because the Device Interface points to the PDO, and the security descriptor on the PDO allows all applications read and write access.

Once again recall that – once the application has successfully opened some Device Object in the device stack -- all I/O operations from the application will always go to the top of the device stack. So, once again, requests are sent to the FDO regardless of whether the PDO or FDO was used.

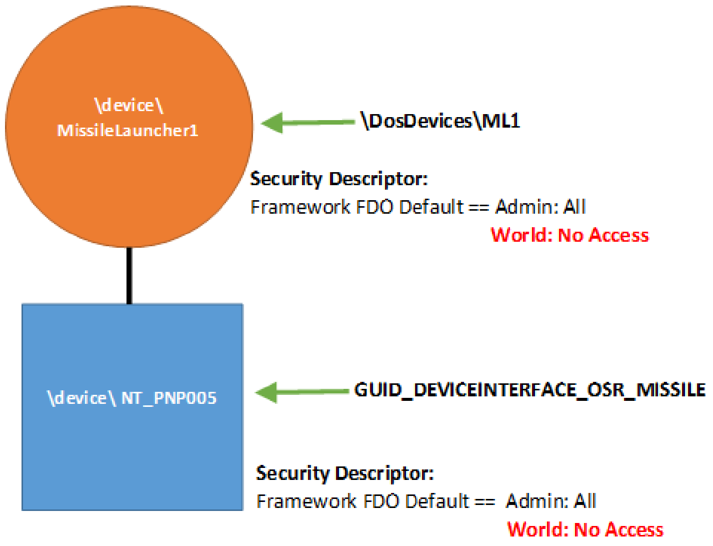

\device\ MissileLauncher1
\DosDevices\ML1
Security Descriptor:
Framework FDO Default == Admin: All
World: No Access
\device\ NT_PNP005
GUID_DEVICEINTERFACE_OSR_MISSILE
Security Descriptor:
Framework FDO Default == Admin: All
World: No Access

Figure 5—*Named* FDO, Device Interface Specified, Device Setup Class Specified

Another example? In this case, we’ll do exactly what we did in the example shown in Figure 4 (named FDO, symbolic link created to FDO, device interface specified, no last-chance security descriptor provided) but this time we will specify a Device Setup Class by calling **WdfDeviceInitSetDeviceClass**. The results are shown in *Figure 5*.

Note that in Figure 5, the security descriptor that the Framework provided for the FDO is now harmonized across the device stack, and has been applied to the PDO. This ensures that applications encounter the same security descriptor when accessing the device, regardless of whether they access the device via the provided symbolic link name or the device interface.

Let’s look at one more example. In this case, we’ll do exactly what we did in the example shown in Figure 5, but we’ll assume a security descriptor for the Device Setup Class has been stored in the registry. This security descriptor could have been specified during device installation via the INF file, or subsequently by the system administrator. The security descriptor happens to specify all access to administrators, and read access to everyone else.

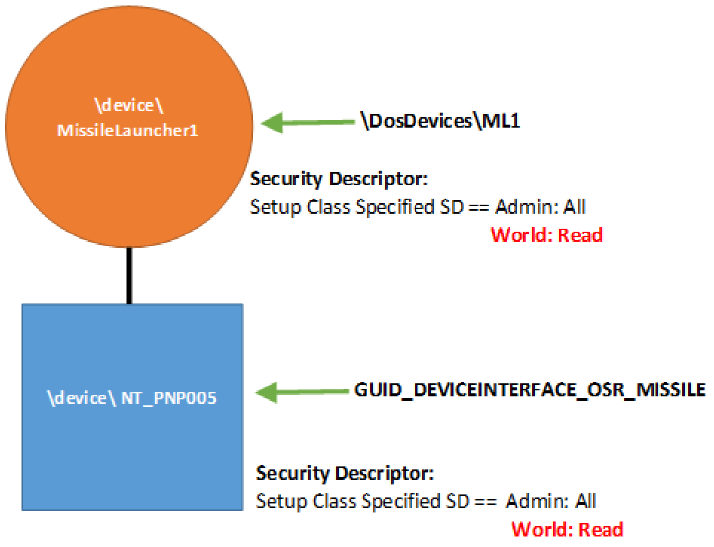

\device\ MissileLauncher1
\DosDevices\ML1
Security Descriptor:
Setup Class Specified SD == Admin: All
World: Read
\device\ NT_PNP005
GUID_DEVICEINTERFACE_OSR_MISSILE
Security Descriptor:
Setup Class Specified SD == Admin: All
World: Read

Figure 6—*Named* FDO, Device Interface Specified, pre-defined security descriptor in the Registry

The resulting device stack is shown in *Figure 6*.

The key thing to notice in Figure 6 is that *the security descriptor is the same on both the FDO and PDO*. This is because Windows ensures that any security descriptor that is associated with a Device Setup Class is applied uniformly across the device stack. Note that, when there is a security descriptor present in the registry for your device or for your device’s Device Setup Class, this harmonization happens even if your driver hasn’t called **WdfDeviceInitSetDeviceClass**. It doesn’t hurt to specify your Device Setup Class, but Windows already knows it and applies the Device Object protection consistently across the device stack.

And again, whether an application choses to open the device via its symbolic link name or its Device Interface, the result is that the security check is consistent. And recall it makes no difference in terms of processing I/O requests which Device Object the application opens. In all cases, I/O operations from the application will be sent to the top of the device stack.

(CONTINUED ON PAGE 32)

Making Device Objects Accessible... (Cont.)

(CONTINUED FROM PAGE 31)

So What Have We Learned?

At a very minimum, I hope that you learned that this topic is more complex than it might first appear. There are also some "best practice" guidelines that I think we can confidently state as a result of our discussion of this topic:

- Don't provide an internal name for your FDO unless your device needs to be found using that name by another kernel mode module, and you can't take the chance the other kernel-mode module will use some other method (such as finding your device by Device Interface GUID).

- If you want user-mode applications to be able to open your device by name, it's perfectly fine to create a symbolic link. Be aware that where that symbolic link points will vary, depending on whether you've provided an internal name for your FDO.

- Creating a Device Interface GUID for your device allows many convenient options for accessing your device, from both user-mode and kernel-mode. You can both create a symbolic link and provide a Device Interface GUID for your device to maximize your user's options. The Device Interface GUID will always point to the PDO.

- Specifying a Device Setup Class when you create your FDO is a best practice, because doing so will cause the protections applied to your device stack to be harmonized if there is no default security descriptor specified in the registry for your device or your Device Setup Class.

- Do not specify a security descriptor for your FDO from within your driver. That is, do not call **WdfDeviceInitAssignSDDLString**. Where and if this security descriptor is used depends on too many other factors. If you need specific protection for your Device Object, specify it in your INF file.

- If you need to set a specific protection for your device, absolutely the best place to do that is in your INF file. You can specify a protection for your specific device or for all devices in your Device Setup Class.

Now you see why what we thought would be a simple article, on a simple topic, became a much more involved examination of device availability and protection. Follow the guidelines, and you can't go wrong! **[Again, See *Specifying Security In The INF File*, next page]**

Follow us!

Making Device Objects Accessible... (Cont.)

Specifying Security In The INF File

Your first and best option for specifying protection for the Device Objects in your device stack is using the INF file. Note that an end-user will not be able to hack the INF file and change the protection you specify, as long as your driver install package is signed.

When you specify access controls in your INF, the protection that you specify will be propagated throughout the device stack as described in the larger article. Thus, when you specify protection in your INF, that same protection will be applied to the PDO, FDO, and all filter driver Device Objects that appear in your device stack.

INF Class-Wide Access Controls

Probably the most common way that a specific security descriptor (SD) is set on a device stack is by specifying a default security descriptor for all device in your device's Device Setup Class. This is specified in the INF that defines the device install class via the Security value in the **addreg** section pointed to from the **ClassInstall32** section. Here's an extract from an INF file that defines the OsrExample install class and specifies a default security descriptor for the class:

```
[Version]
Signature="$WINDOWS NT$"
Class=OsrExample
ClassGuid={cab15040-5cc7-11d3-b194-0060b0efd4fd}
Provider="OSR Open Systems Resources, Inc."
DriverVer=2/13/2017,7.1.2
catalogfile=wdfdio.cat

[ClassInstall32]
Addreg=OsrHwClass

[OsrHwClass]
HKR,,,,%ClassName%
HKR,,Icon,,"-5"
HKR,,Security,,"D:P(A;;GA;;;SY)(A;;GA;;;BA)" ;System and Admin only access
```

As previously described, the security descriptor supplied in the INF is defined using Security Descriptor Definition Language (SDDL). When you specify an SD for your Device Setup Class in your INF, the security descriptor is stored in the registry, in the Security value of the Properties key under the **software** (A.K.A. driver) key for your driver. Your device's software key will be:

```
HKLM\SYSTEM\CCS\CONTROL\CLASS\class-guid\instance
```

Look under this key for the key named "Properties." You will need to change the access to the Properties key to be able to see the value named Security. Yes, this is true even if you're an administrator on the box.

INF Per-Device Access Controls

If the device for which your driver is being installed requires different access controls from those specified for your installation class, you can specify a per-device security descriptor in your INF file. A per-device security descriptor is specified in the **addreg** section invoked from the **ddinstall.HW** section of your INF. Following is an extract from an INF file that defines a security descriptor for a particular device within a larger class. You notice that this security descriptor is also defined using SDDL:

```
[MfgDeviceSection]
%DeviceDesc% = WdfDio, PCI\VEN_135E&DEV_8008&SUBSYS_8008135E&REV_01
%DeviceDesc% = WdfDio, PCI\VEN_135E&DEV_8018&SUBSYS_8018135E&REV_01

[WdfDio]
CopyFiles=@WdfDio.sys
```

(CONTINUED ON PAGE 34)

Making Device Objects Accessible... (Cont.)

Specifying Security In The INF File (Cont.)

(CONTINUED FROM PAGE 5)

```
        [WDFDIO.HW]
        addreg=DIOSD

        [DIOSD]
        HKR,,Security,,"D:P(A;;GR;;WD)(A;;GA;;BU)(A;;GA;;;SY)(A;;GR;;;WD)"
```

When you specify a per-device security descriptor in your INF, it is stored in the Registry, in the Security value of the Properties key of your device's hardware (A.K.A device) key. Your device's device key will be:

```
        HKLM\SYSTEM\CCS\ENUM\enumerator\device-id
```

Again, you'll need to change the protection on the Properties key to be able to view this entry.

Note that specifying per-device access controls in your INF overrides *for your device stack only* any class-wide access controls that might have been specified when the device class was defined. This is true regardless of whether the security descriptor you supply is more or less secure than the default protection. Thus, specifying a per-device security descriptor for a device allows you to specify precisely the protection that your device should have, without affecting the devices created by any other drivers in the class.

Best Practices... (Cont.)

(CONTINUED FROM PAGE 3)

Build your solution with warning level /W4 /WX
Why: Sure, there are warnings that you might want to globally suppress. But the header files should all build cleanly with /W4 now, and the more checking your code gets, the less likely you are to have latent errors. And if you're going to jump in and build with all warnings, you might as well treat warnings as errors to keep your build clean.

Enable Code Analysis (CA) for every build of your driver project. Enable All Rules.
Why: We like to think of Code Analysis as a super-smart extension of the compiler warnings. Again, anything that helps file bugs for you is good.

Testing

Enable Windows Driver Verifier for your driver during all testing. Just turn it on and leave it on. On your test machine in your office, and on your test machine in the lab. Enable everything except the various Low Resource features.
Why: Being able to run your driver successfully under Driver Verifier is *the* only way to have confidence that your driver is properly written. Because it never (well, almost never) causes noise or false-positives, it's the kind of thing you can just enable and forget about.

Enable Windows Driver Verifier for any wrapper/library modules you use. For example, if you're writing a KMDF driver, be sure to enable to enable Driver Verifier on the Framework. If you're writing a StorPort MiniPort, enable Driver Verifier on StorPort as well as your own driver.
Why: When you use a "wrapper" or "support library" much of the work your driver does is done on your behalf by that "wrapper" or "library" – you absolutely need that work monitored by Driver Verifier just as if your driver had done it. Note that you can also enable Driver Verifier on the kernel itself. This, for example, causes allocations made by the kernel to be subject to Special Pool.

Test on the checked build of the OS. Test on a fully checked build if you can find it (and you can successfully get it to install), test on a partially checked build otherwise. Checked kernel and HAL images are provided as part of the WDK.
Why: Checked build asserts are added by the developers of the OS code to validate the assumptions they are making in their code. The checks find unique problems that you would never find otherwise.

Run the Hardware Lab Kit (HLK) tests on your driver. Even if you don't currently need logo certification, just get the environment set up and start running the tests periodically.
Why: The HLK tests can either be very minimal (e.g. for Unclassified devices and drivers) to intensive and sometimes seemingly arbitrary (e.g. the File System Filter driver tests). The tests attempt to determine if the system behaves differently with your driver present than without. Better to get a feel for this periodically than to stay blissfully unaware and get a rude awakening in the future.

Run SDV Periodically Before Release, if your driver model is supported. Enable All Rules.
Why: It took us years to be able to say this but… SDV actually finds bugs. There. It took years, but SDV is actually a damn useful tool, especially when coupled with SAL.

Test Exactly What You Ship. This should go without saying, but… Before releasing your product to the world, be sure that you replicate the "real world" scenarios of how your product will be built and installed. For example, before shipping your driver, you'll want to test at least once *without* Windows Driver Verifier enabled and *without* WinDbg attached to the system. Test what your customers will use.
Why: If you don't test what your customers will ultimately use, you run the very real risk for shipping something that does not work. We learned this the hard way.

Follow us!

OSR Seminar Schedule

Seminar	Dates	Location
Internals & Software Drivers	27-31 March	Dulles/Sterling, VA
WDF Drivers I: Core Concepts	15-19 May	**At OSR!** Amherst/Nashua, NH
WDF Drivers II: Advanced	23-26 May	**At OSR!** Amherst/Nashua, NH
Kernel Debugging & Crash Analysis	26-30 June	Dulles/Sterling, VA
Internals & Software Drivers	24-28 July	**At OSR!** Amherst/Nashua, NH
Developing File Systems Mini-Filters	TBD	TBD

More Dates/Locations Available—See website for details

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...

"This was the third time I took an OSR course. For Windows kernel mode learning, I wouldn't go anywhere else. Even if I need to fly to the other side of the globe, I'd still go with OSR."

-Recent attendee of OSR WDF Core seminar

THE NT INSIDER - You Can Subscribe!

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

Join OSRHINTS

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.