ISSUE VOLUME
02/23
SEPT-OCT
2017

The NT Insider

A publication of OSR Open Systems Resources, Inc.

```
MODULE_AUTHOR("OSR Open Systems Resources, Inc.");
MODULE_DESCRIPTION("The NT Insider");
MODULE_LICENSE("GPL");

static struct file_system_type ntinsiderfs_fs_type = {
        .owner           = OSR,
        .name            = "The NT Insider",
        .fs_flags        = FS_SEPTEMBER | FS_OCTOBER | FS_2017,
        .mount           = ntinsiderfs_mount,
};

static int __init init_ntinsiderfs_fs(void)
{
        int err;

        err = init_sept_oct_issue();
        if (err)
                panic("FESF on Linux!", 3);

        err = we_need_a_driver_developer_conference(4);
        if (unlikely(err))
                goto out_uninit_issue;

        err = handling_cleanup_close_cancel(6);
        if (err)
                goto out_destroy_pontification;

        standard_and_isolation_minifilters = 8;

        /*
        * TODO: Read about the Isolation Minifilter Solution Framework (IMSF)
        * for Windows on Page 10
        */

        the_scoop_on_universal_drivers = 12;

        err = register_filesystem(&ntinsiderfs_fs_type);
        if (likely(err))
                goto out_cleanup_cleanup_close_cancel;

        return 0;

out_cleanup_cleanup_close_cancel:
        cleanup_cleanup_close_cancel();
out_destroy_pontification:
        destroy_peter_pontificates();
out_uninit_issue:
        uninit_sept_oct_issue();

        return err;
}

module_init(init_ntinsiderfs_fs);
```

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

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

Stuff Our Lawyers Make Us Say

All trademarks mentioned in this publication are the property of their respective owners. "OSR", "OSR Online" and the OSR corporate logo are trademarks or registered trademarks of OSR Open Systems Resources, Inc.

We really try very hard to be sure that the information we publish in *The NT Insider* is accurate. Sometimes we may screw up. We'll appreciate it if you call this to our attention, if you do it gently.

OSR expressly disclaims any warranty for the material presented herein. This material is presented "as is" without warranty of any kind, either expressed or implied, including, without limitation, the implied warranties of merchantability or fitness for a particular purpose. The entire risk arising from the use of this material remains with you. OSR's entire liability and your exclusive remedy shall not exceed the price paid for this material. In no event shall OSR or its suppliers be liable for any damages whatsoever.

It is the official policy of OSR Open Systems Resources, Inc. to safeguard and protect as its own, the confidential and proprietary information of its clients, partners, and others. OSR will not knowingly divulge trade secret or proprietary information of any party without prior written permission. All information contained in *The NT Insider* has been learned or deduced from public sources...often using a lot of sweat and sometimes even a good deal of ingenuity.

OSR is fortunate to have customer and partner relations that include many of the world's leading high-tech organizations. As a result, OSR may have a material connection with organizations whose products or services are discussed, reviewed, or endorsed in *The NT Insider*.

Neither OSR nor *The NT Insider* is in any way endorsed by Microsoft Corporation. And we like it that way, thank you very much.

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:

Tracking an NTSTATUS to its Source
I found myself in a situation this week where I really wanted to call the API SeTokenIsAdmin. I vaguely remembered some issues around this API, and Googling quickly brought up a couple of threads from NTDEV and NTFSD hinting at a security issue that was fixed in 2015...
http://www.osr.com/blog/2017/07/14/tracking-ntstatus-source/

Attestation Signing — It's NOT a Mystery
Over the past few weeks, we've been contacted by several folks — clients and non-clients alike — with a variation on the above story, in various levels of panic. What they have in common is that they're all absolutely mystified by the process that's required to get their drivers to install under certain circumstances on newer versions of Windows.
http://www.osr.com/blog/2017/07/06/attestation-signing-mystery/

WinDbg, Debugger Objects, and JavaScript! Oh, My!
In case you've missed it, there are tons of changes going on under the covers in WinDbg. There is a fundamental paradigm shift going on in terms of how WinDbg grants access and presents data to the user and it can lead to some pretty cool results.
http://www.osr.com/blog/2017/05/18/windbg-debugger-objects-javascript-oh/

1394 Boot Debugging is Dead
TL;DR: Don't waste your time like we did – 1394 boot debugging no longer works on the latest builds on Windows 10.
http://www.osr.com/blog/2017/04/07/1394-boot-debugging-dead/

Is Running the HLK Tests REALLY a Best Practice?
It turns out that how useful the HLKs are depends almost entirely on the type of driver your writing.
http://www.osr.com/blog/2017/03/22/running-hlk-tests-really-best-practice/

Visual Studio 2017 Released — Driver Devs: Stay Where You Are
http://www.osr.com/blog/2017/03/07/visual-studio-2017-released-driver-devs-must-stay-vs-2015/

Of Windows, Git, FUSE, and Moral Equivalence
It's a known fact these days that Microsoft is feeling the Git love...
http://www.osr.com/blog/2017/02/24/windows-git-fuse-moral-equivalence/

Unexpected Case of Bugcheck IRQL_UNEXPECTED_VALUE (C8)
Yet another interesting case lands on our doorstep thanks to NTDEV...
http://www.osr.com/blog/2017/02/17/unexpected-case-bugcheck-irql_unexpected_value-c8/

Windows Compatible Enterprise Encryption Comes to Linux

File Encryption Solution Framework for Linux

Maybe you're wondering if you're truly reading *The NT Insider*. Is this really an article on *Linux*? Well, yes and no. For years, we've been working on and writing about our on-access transparent File Encryption Solution Framework (FESF). This Solution Framework provides everything you need to build your own custom fast, reliable, file encryption product with no kernel-mode development required. We provide all the kernel-mode code, including an Isolation Minifilter and all the supporting code that performs the actual encryption using the Windows CNG system. Plus, we provide a complete user-mode reference implementation (written in C++) that you can use to start your project.

You define the policy that determines which files get encrypted or decrypted on access, based on the parameters you establish: The accessing application, the path or name of the file being accessed, the user or group doing the access, or just about any other factor you can envision.

We've had clients take FESF and go to market with their own complete, finished, product in less than six months. We think that's pretty remarkable.

Not Everybody Uses Windows!
Yup... it surprised us too. OK, it didn't surprise *everyone* at OSR but, it surprised a *lot* of us. Well, maybe not really a *lot* of us, but... at least it surprised Peter. Anyhow, indeed not everyone on the planet uses Windows. There are *actual people* who run Linux on their servers and workstations. And, not surprisingly, these folks need on-access transparent file encryption.

While there are a couple of good quality transparent file encryption solutions for Linux already, none of them has anything near the flexibility that FESF provides. For example, they don't allow you to specify that you only want to encrypt *.odt files in a given directory. Or that you only want those *.odt files transparently encrypted/decrypted when they're opened with LibreOffice Writer (as opposed to say, when they're accessed using catdoc).

And, of course, none of the existing Linux data encryption applications are compatible with FESF for Windows.

Introducing: FESF for Linux
OSR's File Encryption Solution Framework (FESF) for Linux provides you with the ability to create your own enterprise grade, highly flexible, transparent on-access file encryption/decryption solution on Linux. FESF for Linux comprises a native (kernel-mode) layered file system, written from the ground-up to support file access interception and file encryption activities. Using FESF for Windows as its inspiration, FESF for Linux allows your product to make its policy decisions in real-time from user-mode. Using FESF for Linux you can create a customized, powerful, and flexible transparent data encryption product with no kernel-mode programming. FESF for Linux comes with user-mode code that you can use as the basis for your product, giving you a head start on your development effort.

(CONTINUED ON PAGE 27)

Peter Pontificates

We Need a Driver Developer Conference!

Back in 2015, I blogged and pontificated at great length about how I welcomed the improved openness and communication that seemed to be happening between Microsoft and the Windows driver development community. Of course, things could hardly have gotten worse than they were during the era that took us from Windows 7 to Windows 8. Oh, the stories I could tell. For example, I remember a major IHV (total sales in the realm of US$15 billion) that happened to be a client of ours being told that Microsoft wasn't interested in engaging with them, and if they wanted to know what was going on in Windows they could ask the OEMs to whom they sold their hardware.

Ouch! Things could only improve from there, right?

Well, they DID improve from there. Radically. And, thankfully, those days are long over. Still, even at the dawn of the new *glasnost* in 2015, I was calling for even more and better two-way communications between Microsoft and us third party driver devs. And almost all the points I made in my December 2015 Pontification calling for Even Better Communication still stand today.

Starting with Windows 8 and on through Windows 10, there have been tons of changes in the Windows ecosystem. There have been some key architectural changes to both the I/O and Power Management subsystems. We've witnessed the introduction of Windows RT, the introduction of Windows Mobile, the death of Windows Phone, the introduction of Windows IoT Core (on x86 and ARM), support for the Raspberry PI, support for the Snapdragon 835, introduction of OneCore, support for x86 emulation on ARM, and the introduction of Windows 10 S. And that's just to name a few of the most interesting things that have happened in the last couple of years.

Oh, I almost forgot to mention: In this same time period, the C/C++ compiler that's part of Visual Studio was largely re-written. And something called Universal Drivers was introduced (if you haven't grasped this, it's a biggie). And changes to INF files have been recommended. And who can miss the changes that have taken place to driver signing?

Should Microsoft Hold a Driver Developer Conference?

VOTE in our 6-question online poll: https://www.osr.com/microsoft-hold-conference-driver-developers/

You'll be able to see the live results after you vote, and we'll summarize all the results in the next issue of *The NT Insider*.

Let's hear your voice!

Vote Now!

What's notable to me is that *all the changes I mentioned, plus many others,* ***significantly*** *impact driver developers*.

What's even **more** notable to me is that there have been few opportunities for driver developers to come up to speed on these changes. It's definitely **not** like is was back in the bad old days. Most of the information on these topics is "out there" **somewhere**. Much has been documented on MSDN or on a blog. There are Channel 9 videos on several of these topics. Heck, we've written articles here in *The NT Insider* on a number of these topics as well.

But, by and large, driver devs have had to find their own way when it came to figuring out what's new for Windows driver developers. They've had to discover, search out, identify, and attempt to consume information about each topic. They read an article here, they watch a video there. And they've had little help piecing together the bigger picture.

(CONTINUED ON PAGE 5)

Peter Pontificates... (Cont.)

(CONTINUED FROM PAGE 4)

How has Microsoft gotten information to driver devs at other times in Windows history? In years past, Microsoft has held periodic conferences to educate driver and hardware developers. At various points in history these conferences have been WinHEC, a Driver Developer Conference, Microsoft Build, or some other event.

For the past few years, Microsoft has held WinHEC in Greater China. In March of 2015 there was a WinHEC conference in Shenzhen, in 2016 there were WinHEC Summer Workshops in both Shenzhen and Taipei in June/July and a full WinHEC conference in December. The most recent WinHEC event was in Taipei in June of this year.

Look: I get the Microsoft has ignored developers all throughout Asia forever. I agree that a lot of ODMs are located in Greater China. I understand, and agree, that by the time 2015 rolled around it was long past time for Microsoft to hold an event like WinHEC in Greater China.

According to Microsoft:

> *"The Windows Hardware Engineering Community (WinHEC) is where technical experts from around the world, and Microsoft, come together to make Windows great for every customer. The forum is designed to help educate, facilitate the exchange of ideas and give people a venue to share best practices and discuss future opportunities."*

But for three years in a row? Seriously? Shenzhen is on the other side of the planet to everyone BUT the Greater China development community. I've been to Shenzhen. It's a pretty cool place. The food is amazing. And it's just a short ride from Hong Kong**. But it's no place for Microsoft to exclusively present content that's important to driver developers.** If WinHEC is "where technical experts from around the world, and Microsoft, come together to make Windows great" then WinHEC needs to travel. If the idea is to "help educate [and] facilitate the exchange of ideas and give people a venue to share best practices" then WinHEC can't JUST be held in Greater China.

I think it's time -- well, long past time, actually – for Microsoft to have a conference for driver developers. I don't care if it's WinHEC (where there's traditionally been a ton of info shared that's been directed to OEM platform planning type people) or it's like the previous Driver Developer Conference. But it needs to focus specifically on getting information to driver devs. I don't want a few presentations thrown into some other event. We don't need that.

In discussing this idea recently with folks out in Redmond, they were certainly open to the idea. They seemed to "get" that we're feeling underinformed.

What do you think? Should Microsoft hold a conference for driver developers? Should it be part of WinHEC or separate? And, if such a conference is organized, where would you like to see it be held?

Vote in our 6-questions online poll: https://www.osr.com/microsoft-hold-conference-driver-developers/
You'll be able to see the live results after you vote, and we'll summarize all the results in the next issue of *The NT Insider*. Let's hear your voice!

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.

Handling Cleanup, Close and Cancel in your WDF Driver

Most of the WDF topics I write about suggest themselves to me as a result of a set of exchanges on our NTDEV list, or as a result of trends I see in driver code that I review or update. This time, I was inspired by both of these.

There was a recent, and reasonably interesting, thread on NTDEV about the use of Cleanup and Cancel Event Processing Callbacks. Also recently, I had the opportunity to update two very different device drivers. One was a WDM driver that was written in the NT V4 timeframe. The other was a KMDF driver written by a new Windows driver developer shortly after the release of KMDF. While these two drivers were for very different types of devices, they had one thing in common: The developers of both misunderstood how to terminate in-progress Requests and used (or, rather, misused) Cleanup events in place of Cancel.

That's how I came to be motivated to write at length about Cleanup, Close, and Cancel handling in WDF drivers. An additional, private, motivation is that while we cover this topic in our Writing WDF Drivers seminar, we almost never have the time to talk about it in the level of depth that I'd prefer. So, I figured an article would be useful for many purposes.

Some Preliminaries

The first thing to understand is that, at least in the driver world (as opposed to the file systems), you need to consider processing for Cleanup, Close, and Cancel together. This is because they're often confused and processing for these functions often interact.

Also, as we begin our discussion, please keep in mind that what we say here about Cleanup, Close, and Cancel is entirely specific to WDF. As in most things WDF, it is never a good idea to attempt to "map" from WDM to WDF (or vice-versa) in your head and expect to be able to figure things out. That way madness lies. The WDF Framework is its own I/O processing ecosystem. Sure, it interacts with WDM, but the rules for how WDF drivers work are established solely and entirely by WDF. Just because something works a certain way in WDM is no reason to believe the same is true for WDF.

The Problem

Let's start with a couple of simple examples of the problems that Cleanup, Close, and Cancel are designed to solve. I want to emphasize that these are just two random examples in which these operations can be useful. There are lots of other important uses for Cleanup, Close, and Cancel events that these two examples don't address.

Assume your driver has a single Queue that uses Sequential Dispatching. A user-mode application successfully calls the Win32 function **CreateFile** specifying FILE_FLAG_OVERLAPPED to allow it to do asynchronous I/O. The app then successfully calls ReadFile 100 times from within a loop, specifying an OVERLAPPED structure each time. The result is that when the application's loop is complete, there are 100 Read Requests pending in your driver.

Now let's say that application calls **CloseHandle** on the handle it used to send those 100 reads to your device. At least some of those 100 reads are still in progress. *What happens to the in-progress I/O Requests? How does your driver handle this situation?*

As a second example, consider what happens when instead of closing the handle to your device when those 100 reads are in progress, the application exits or is terminated. As Windows begins termination processing for the app, your driver potentially has a pointer to one or more data buffers into which to return data into the application's address space. You can see this in *Figure 1*.

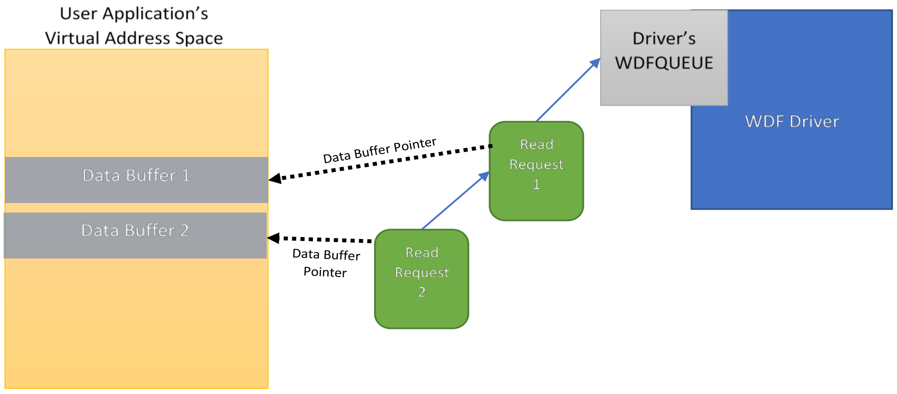


Figure 1 -- WDF Driver with 2 Queued Reads;
Each Read Has a Pointer to a User Data Buffer

In *Figure 1*, you can see a driver with a single WDFQUEUE that has two Read Requests queued (I didn't want to put all 100 I/O Requests in the diagram). Each Read Request has (stored internally) a pointer to the data buffer into which the Read (output) data is to be returned.

If Windows were to allow the Process to terminate completely with I/O requests in

(CONTINUED ON PAGE 7)

Cleanup, Close and Cancel... (Cont.)

(CONTINUED FROM PAGE 6)

progress, and if it were to free the physical memory the process was using including the application's data buffers, the driver would have the potential of writing on random memory and corrupting the system. That would be a very bad thing. As a result, *Windows will not allow the process to exit completely until its I/O operations are completed*. This prevents a driver from unintentionally corrupting memory by processing an ordinary Read Request. As a result, the behavior you see from the application is that the application won't exit, and may even appear to hang, until the pending Requests are completed.

So, when an application exits or is terminated, the question once again becomes: *What happens to the in-progress I/O Requests? How does your driver handle this situation?*

Cleanup, Close, and Cancel Defined

The two problems we just described, closing a handle with I/O in progress and an application attempting to exit with I/O in progress, are two of the primary issues that Cleanup, Close, and Cancel are designed to manage. Let's start with some basic definitions of each of these operations. In the next section, we'll expand on these definitions and describe more about the specifics of how your driver can handle the operations.

Event Processing Callbacks for Cleanup, Close, and Cancel are all technically optional in WDF. That means, your driver will need to specifically opt-in to handle them. Cleanup and Close are File Object specific, so Event Processing Callbacks for these operations are related to WDF File Object handling. If, as you read this section, you feel like you need a quick refresher about Windows File Objects, see the sidebar **About File Objects** (below).

Let's start by discussing Cleanup. When the last handle to a File Object is closed, Windows issues a Cleanup request and WDF calls the associated driver's *EvtFileCleanup* Event Processing Callback function. Because there's almost always only one handle associated with a given File Object, as soon as an application calls **CloseHandle** (and synchronously from within that function call), WDF will call the driver's *EvtFileCleanup* Event Processing Callback. *EvtFileCleanup* is always called in the context of the thread/process calling **CloseHandle**, and at IRQL PASSIVE_LEVEL. For a description of the unusual case when an apps call to CloseHandle may NOT result in an immediate Cleanup operation, see the sidebar entitled **About File Objects**.

(CONTINUED ON PAGE 20)

About File Objects

A File Object in Windows – the WDM data structure named FILE_OBJECT and the WDF Object WDFFILEOBJECT – represents a single, specific, open instance of a file or device. For example, when an application successfully calls CreateFile, a File Object is created that represents that open instance. If the call to CreateFile is successful, it returns a handle to this File Object, that the application uses to perform subsequent I/O operations on the file or device. The returned handle is a process-specific reference to a particular File Object.

The File Object is the general I/O Subsystem object that is used to keep track of and manage I/O operations on a given handle. For example, when you issue sequential read operations to a file on disk without specifying a specific file offset, the file system uses the File Object to keep track of the next offset in the file to read. Each File Object is almost always associated with only a single handle. There are, however, some relatively rare cases (such as when the Win32 function DuplicateHandle has been used) when there can be more than one handle associated with a single File Object.

You can read more about WDF File Objects, their properties, and the Event Processing Callbacks associated with them, in The NT Insider article WDF File Object Callbacks and Properties Demystified.

Each time the application issues an I/O request using the File Object's handle, the reference count on the File Object is incremented. Each time a driver or file system completes an I/O operation on a given File Object, the File Object's reference count is decremented.

File System Minifilters

An Introduction to Standard and Isolation Minifilters

The filter driver concept is one of the most powerful architectural features of the Windows I/O subsystem. A filter can add value to the functionality of an existing device by simply attaching itself over that device. And, of course, filtering a device requires no change to the driver for the underlying device.

Filter drivers are installed at many levels in a typical Windows system. For example, there are standard Windows-supplied filter drivers at the volume level that provide volume snapshot functionality (to support backups), as well as optional full volume encryption (in support of Windows Bitlocker). There may also be manufacturer-specific filter drivers, such as a filter for a mouse or keyboard that adds support for buttons that are unique to a given model.

One of the most common, and also the most powerful, places to insert a filter in a Windows system is over a file system. File system filters intercept I/O operations (from both applications and the system itself) before those I/O operations reach the file system. This allows them to monitor, track, manage, manipulate, and even accept or reject I/O operations before the file system gets to see them. The type of file system filter that most people are familiar with is probably the antivirus filter. This type of filter typically intercepts file open requests and suspends them while the filter (or, more likely, an associated service running in user mode) scans the file being opened for viruses. If any viruses are found, the open request can be canceled. If no viruses are found, the open request can be allowed to complete normally.

File system filters are commonly used for everything from antivirus and malware scanning as just described, to software license tracking and management, to auditing and changed tracking on files, to on access transparent data encryption and decryption. File system filters can also be used for other, less obvious, purposes. For example, because they see which files are created and written, file system filters are often play key roles in backup products and hierarchical storage subsystems. And because file system filters are able to be the first interpreters of the file system “name space” that applications see, they can also perform powerful file redirection operations, such as making a remote file (such as one stored somewhere in the cloud) appear to be local.

Since its introduction in Windows XP SP2, the File System Minifilter model has become the preferred mechanism for implementing file system filters. This is for good reason, because the Minifilter model provides an excellent organization and support framework for file system filter driver development. With reasonably good documentation and a significant set of examples on GitHub, many devs feel that writing a file system Minifilter is well within their ability. And they’re right… provided they stay within certain boundaries.

This article describes the basic architectural concepts of Windows file system Minifilters. It then describes two distinct types of Minifilters: **Standard Minifilters**, and **Isolation Minifilters**. Finally, it describes why a project to develop a file system Isolation Minifilter is significantly more complex than a project to develop a Standard Minifilter.

Standard Minifilters vs. Isolation Minifilters

In this article, we clarify a couple of common terms that are in use in the Windows file system community.

Standard Minifilter
A Standard Minifilter is a Windows file system Minifilter driver that monitors or tracks file system data. Most all antivirus scanners are Standard Minifilters.

Isolation Minifilter
An Isolation Minifilter is a Windows file system Minifilter driver that separates the view(s) of a file’s data from the actual underlying data of that same file. A typical example of an Isolation Minifilter is an on access transparent encryption/decryption filter. Isolation Minifilters use the “same stack” concept, and provide their different views by providing unique cache sections for each view.

Filter Manager and Altitudes
The basis for the file system Minifilter model is Filter Manager, which is a standard Windows component. Filter Manager is implemented as a legacy file system filter, and filters all file system instances. Because there can be multiple filters located over any given file system instance (the default installation of Windows 10 includes no less than nine standard file system Minifilters!), Filter Manager provides a system of “altitudes” that allows a developer to decide where in the filter hierarchy their Minifilter should be installed. See *Figure 1 (next page)*.

In *Figure 1*, you can see Windows Filter Manager (labelled FLTMGR) filtering the instance of the NTFS file system that’s mounted as the C drive. In the figure, Filter Manager has three file system Minifilters loaded: MiniFilter A, MiniFilter B and MiniFilter C. The order of

(CONTINUED ON PAGE 9)

File System Minifilters... (Cont.)

(CONTINUED FROM PAGE 8)

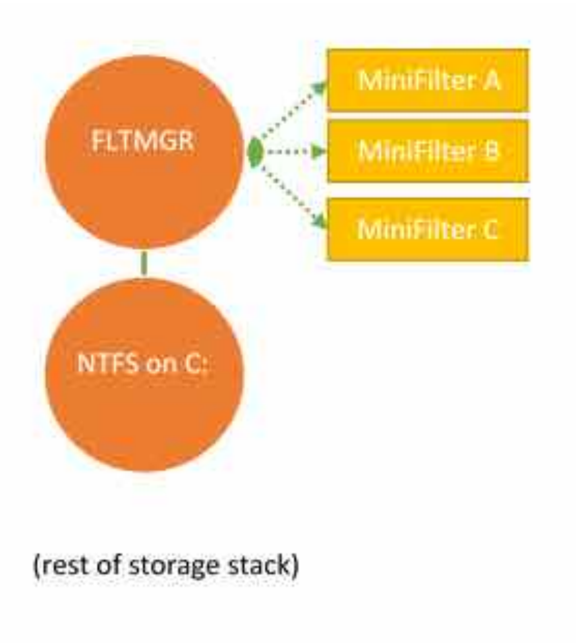


Figure 1 -- Filter Manager with Three File System Minifilters

these filters, MiniFilter A being above MiniFilter B, and MiniFilter B being above MiniFilter C is not random. Rather, this is determined by the Altitude assigned to each of the Minifilters. Altitudes are unique per Minifilter and are specified during Minifilter installation.

Altitude is an important attribute for file system filters, because filtering at the right altitude can be critical to proper operation. For example, consider two Minifilters that might be installed over an arbitrary file system: An on-access transparent data encryption Minifilter and an antivirus Minifilter. In order for the antivirus Minifilter to do its work, it needs to operate on the decrypted contents of files. Therefore, the antivirus Minifilter would need to be higher in altitude (that is, above) the data encryption Minifilter.

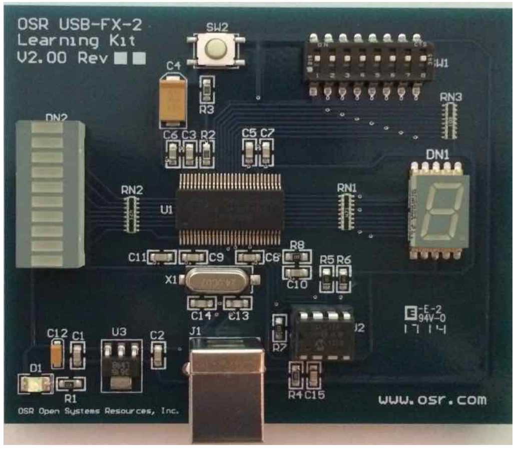
Minifilter Callbacks

When a Minifilter registers with Filter Manager, in addition to other things, it may elect to receive PreOperation and/or PostOperation callbacks for specific I/O operations. PreOperation callbacks are invoked before each I/O operation of the specified type is passed on to the file system being filtered (or, in fact, the next lowest altitude Minifilter). PostOperation Minifilter callbacks are invoked after the file system (and any lower Minifilters) have processed the particular type of I/O operation.

Filter Manager's support for callbacks is very well thought out. For example, when a given Minifilter receives a PreOperation callback, that filter can:

- Complete the operation entirely. This results in lower altitude Minifilters (if any), and even the file system that's being filtered, not seeing this I/O operation.
- Complete the operation, pass it along to lower altitude Minifilters (if any) and the underlying file system, and elect to have a callback when the operation is complete (PostOperation callback).
- Complete the operation, pass it along to lower altitude Minifilters (if any) and the underlying file system, but elect NOT to have a callback when the operation is complete (this is the PostOperation callback).
- Return with the operation in progress. In this case, the Minifilter will inform the Filter Manager later as to the status of the operation, including whether any underlying entities and need to be called or if a PostOperation callback is required.

(CONTINUED ON PAGE 16)



A Hard Challenge Solved
OSR's Isolation Minifilter Solution Framework

Every week, we encounter a problem like this: An unsuspecting dev looks at the File System Minifilter model, sees that it's pretty cool, and decides to undertake what looks like a straightforward project. The project? A Minifilter that does one of the following:

- Virtualizes the location of a file (allowing the file to be pulled down, bit by bit, from back-end cloud storage or from slow near-line storage).
- Implements per-file on-access transparent encryption and decryption
- Implements "hot backup" of files, and allows seamless access to both new and old versions
- Implements file compression
- Provides data deduplication processing (store the file data once, have it be available via multiple files)

The dev gets coding underway, but after a few weeks runs into unexpected issues. Why? Because he or she has left the realm of conventional **Standard Minifilters** and accidentally entered the world of Windows file system **Isolation Minifilters**. We describe the technical differences between Standard Minifilter and Isolation Minifilters in another article in this issue of *The NT Insider*.

Should This Minifilter Be an Isolation Minifilter?

Unless you have experience in Windows file systems or file system filters, it's sometimes hard to know in advance that your design is going to (or should) enter into Isolation Minifilter territory. Heck, here at OSR we've started Minifilter projects, only to discover after several weeks of work that "we probably should have used an Isolation Minifilter for this." It happens.

The best way to determine if you're likely to need an Isolation Minifilter, *before* you start your code, is to ask yourself, "Do I need to maintain a view of the data that is different in content, shape, or location from that which is stored by the underlying file system." If the answer to this question is yes, you should at least consider implementing an Isolation Minifilter.

Isolation Minifilter: Hard

Determining whether you need to write a Standard Minifilter or an Isolation Minifilter is important, because undertaking the development of an Isolation Minifilter is far more difficult than a Standard Minifilter. Because of their close coupling with the Windows VM subsystems (Memory Manager and Cache Manager) developing an Isolation Minifilter is at least as complicated as developing a Windows file system. In fact, in discussing just this topic, one of the OSR devs recently wrote:

> *I maintain that an Isolation Minifilter is actually much harder than a filesystem to write because in many cases the lower edge [of the Minifilter] is a filesystem (which is complicated and involves locking), whereas for a filesystem it is a series of blocks (idempotent, no locking, etc,….)*

(CONTINUED ON PAGE 11)

File System Minifilters... (Cont.)

(CONTINUED FROM PAGE 10)

In any case, let's just agree that Isolation Minifilters are difficult to write from scratch. Based on our own experience developing Isolation Minifilters, starting from scratch we estimate that it would take a very experienced Windows file system developer at least 3 months to get a *prototype* Isolation Minifilter working. For a less experienced dev, one who is "just" experienced writing Windows kernel-mode drivers, we figure it would take at least 6-9 months to get a working *prototype***.** And, in either case, that's just to have something that you can demo. A real solution, that works in the real world, will still be many months of work away.

Developing an Isolation Minifilter

From scratch to prototype:
3-9 months

From OSR's IMSF to prototype:
1-3 weeks

Isolation Minifilter Solution Framework: Much Easier
Fortunately, there's a solution that allows you to jump-start the process of developing an Isolation Minifilter and *have working prototype in a few weeks instead of many months*. That solution is OSR's Isolation Minifilter Solution Framework (IMSF).

As we were developing our comprehensive Solution Framework specific to on-access transparent file encryption (FESF, you can read about it here if you're interested) we, obviously, had to implement an Isolation Minifilter. Instead of tightly coupling this Isolation Minifilter with FESF, we architected and implemented it as a separable set of modules. We make these modules available, with documentation and example code, as part of the IMSF.

Our IMSF kit comprises a complete and fully customizable Isolation Minifilter. This Isolation Minifilter is identical to the code that forms the heart of FESF that is currently shipping world-wide in major products. This code gets used and tested every day, in real-world conditions. It works. It's solid.

In addition to the Isolation Minifilter itself, IMSF includes comprehensive documentation for the interfaces and a working example that illustrates how to use IMSF as the basis for a real product. And, yes: You get the complete source code. For everything.

To be clear: You still need to understand Windows kernel-mode programming and file systems concepts to make good use of our IMSF kit. But the hard work is done for you by IMSF. You don't need to be an expert on the Cache Manager. You don't have to figure out "How should I do this?" IMSF provides a base that's solid, actively developed, constantly maintained with each new release/update of Windows, and is *proven to work* in real, shipping, products.

Ask an Expert
If you're working on a project, and you'd like to discuss whether the Isolation Minifilter approach might be the right way to go, get in contact with us. There's no charge to get our engineering team on Skype with your engineering team to discuss your options. If we don't think you need an Isolation Minifilter, or if IMSF isn't right for you, we'll tell you. On the other hand, maybe we can save you some time and trouble with one of our Solution Frameworks. Either way, if can start you in the right direction with just an hour on the phone, we figure it'll be time well spent.

Follow us!

One Windows Everywhere, One Driver Everywhere

The Scoop on Universal Drivers

If you've been paying attention to the driver space at all since the release of Windows 8, you've probably noticed the term **Universal Driver**. What you may not know is what it means, why you should care, or how the Universal Driver concept is evolving, as the Windows platform and systems it supports evolve.

The Road to OneCore

Back in the days before Windows 8, Microsoft began the task of componentizing Windows both at the OS level and the level of support APIs. It was a long and difficult journey. I'll spare you the details, but an actual, componentized, version of Windows finally made its appearance as part of Win10 1607 (RS1, the Anniversary Update). The componentized OS is referred to as OneCore, and the componentized application support is referred to as OneCoreUAP (Universal Application Platform).

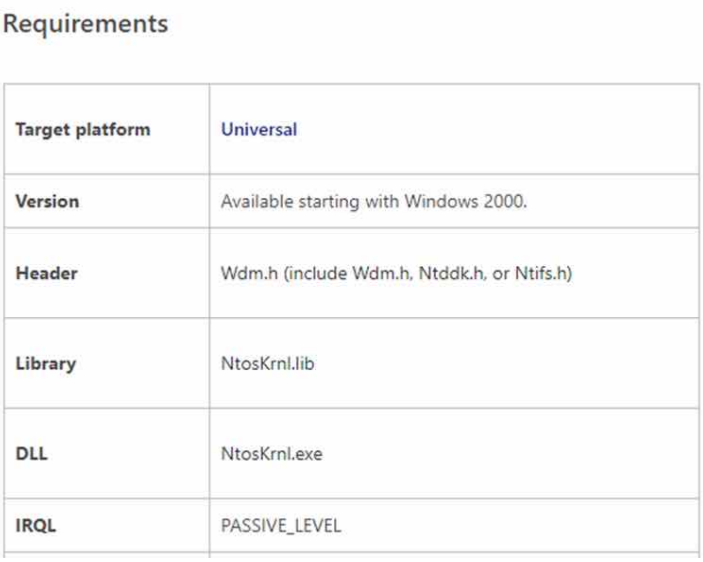
What makes this interesting to driver writers is that, for the first time, there is now a single core Windows component that runs on all manner of platforms: Desktops, Servers, tablets, IoT appliances, Surface Hub, Xbox, and even phones. If you could, ah, *find* a Windows Phone. But that's a different story.

Because of Windows' strong architectural underpinnings, devs can write a single driver that will potentially support a vast array of different hardware. No re-writes, no conditional code. Just retarget, rebuild, and you're good to go.

Sound good? Let's talk about how you can get there.

Universal Driver DDIs

Universal Drivers can be either KMDF, WDM, or UMDF 2 drivers. The primary constraint to writing a Universal Driver using one of these models is that you must restrict the functions your driver calls to those that are part of OneCore (for kernel-mode drivers) or OneCoreUAP (for user-mode drivers). These Device Driver Interfaces (DDIs) and APIs are identified in the MSDN docs as being supported by the "Universal" target platform. See *Figure 1* for an example.



Requirements

Target platform	Universal
Version	Available starting with Windows 2000.
Header	Wdm.h (include Wdm.h, Ntddk.h, or Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

Figure 1 -- Identifying Universal APIs and DDIs; Here's a Kernel-Mode Example

For kernel-mode drivers, sticking to the universal DDIs is easy. In fact, it's pretty hard to find a DDI that's *not* Universal. When you do find one, it's probably not a function you want to call (or should call) anyways. For example, when's the last time you called **KeGetCurrentProcessorNumber**, eh? Well, it hasn't worked correctly since the introduction of Windows 7 (with the introduction of processor groups). Anyhow, it's a DDI that is not Universal. Interestingly enough, even lots of the DDIs that were long-ago relegated to NTDDK.H are part of the Universal DDI set. For example, **MmIsAddressValid** is one of those DDIs that doesn't do what most people think it does, and was moved into NTDDK.H back in the stone age. But, it's a Universal DDI. Yay. I think.

The MSDN docs list a few surprising DDI families as *not* being Universal, such as the Remove Lock set of DDIs (**IoAcquireRemoveLock**, **IoReleaseRemoveLockAndWait**, and their friends). However, the good news is that MSDN appears to be wrong about these. There's a file that's part of the WDK that lists (in XML, for your convenience) all the Universal APIs and DDIs. That file is:

C:\Program Files (x86)\Windows Kits\10\build\universalDDIs\<arch>\UniversalDDIs.xml

In the case of the Remove Lock DDIs, the functions appear in the above XML file even though they're listed in MSDN as not being Universal. The XML file wins. These functions are indeed part of the Universal DDI set.

(CONTINUED ON PAGE 13)

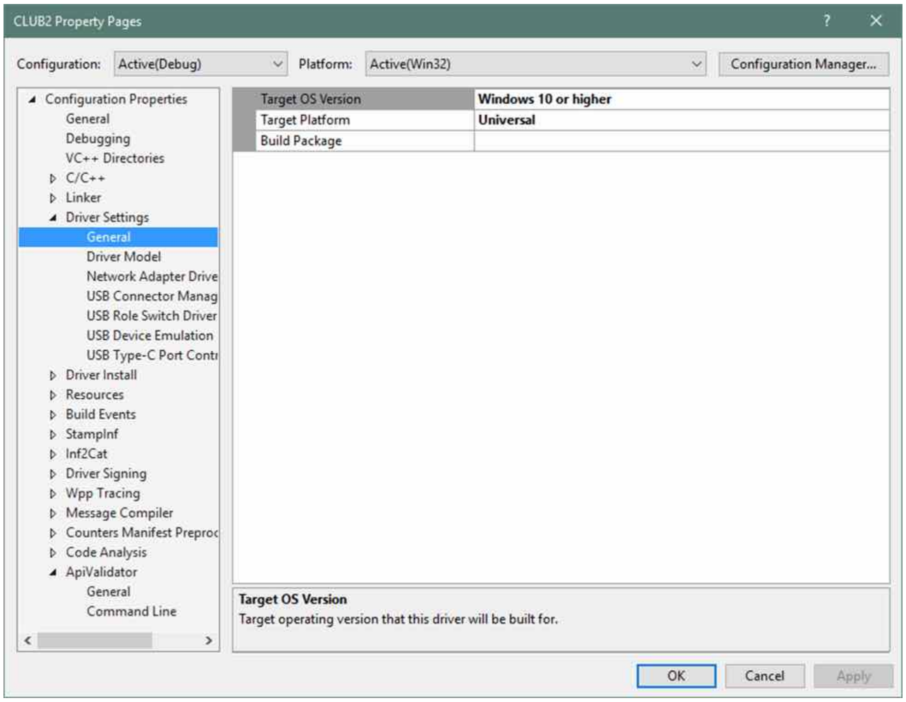
Universal Drivers...(Cont.)

(CONTINUED FROM PAGE 12)

For UMDF 2 drivers, the difficulty of sticking to the Universal set of functions depends on how much Win32 you mix into your user-mode driver. If you stick to the WDF DDIs and a few supporting functions, you'll have no problems at all. But the more "application-type code" you have in your driver, the more likely it is that you're going to have to choose alternate APIs or re-write your code to work in a different way.

Luckily, you don't have to guess, or rely on MSDN, or even look through a long XML file, to check to see that you're only calling Universal DDIs. There's a utility, **ApiValidator**, that's part of the WDK that you can elect to run as part of your build process. This utility checks to ensure that everything in your Solution is appropriately Universal (See *Figure 2*).

To prove that creating a kernel-mode driver that sticks to the Universal set of DDIs is really as easy as I claim it is, I put my own code to the test. I rebuilt a reasonably complex KMDF device driver that I've been working on (two-way simultaneous DMA, several PIO type operations via more than a dozen IOCTLs, about 30 source files, and more than 10K NCSLs) that targets Win7 and later as a Universal Driver for Win10. There were no calls to any DDIs outside of the Universal DDI set.



Configuration: Active(Debug)   Platform: Active(Win32)

Target OS Version	Windows 10 or higher
Target Platform	Universal
Build Package	

Target OS Version
Target operating version that this driver will be built for.

Figure 2 -- Select "Universal" for the Target Platform... Use APIValidator to Check Your Calls

But Wait... There's More: Declarative Install

Building a driver that conforms to the Universal DDI set is good. It gets you most of the way towards true universal compatibility. But it's not *all* you need to do.

Let's say you've written a pretty nice SPB driver and you've successfully targeted the Universal DDI set. Now, let's say, you want to install it on a little single-board computer. Maybe it's a Snapdragon 617 system, maybe it's a board with an Apollo Lake processor. But, in any case, the system is running Windows IoT Core.

You might have a great driver, but users have to be able to install it. Windows IoT core doesn't have a standard Windows Explorer interface. In fact, it can run entirely headless. That means

(CONTINUED ON PAGE 14)

Universal Drivers... (Cont.)

(CONTINUED FROM PAGE 13)

there can't be any installer MSI to run and there can't be a co-installer that needs to be executed. You need what's referred to as a Universal INF file that's restricted to "Declarative Driver Installation."

Declarative Install refers to the ability to install drivers in an entirely offline environment, with no GUI interaction. It allows only a limited number of specific, definitive, and unambiguous operations to install the driver. Microsoft claims that not only does this make it possible to install drivers offline, but it also increases the predictability and supportability of driver installations. If that's true, that'd be great, because, in my experience, driver installs either just work or they're a complete pain in the ass. They never seem to be anything in between. Refer to the MSDN docs on Universal INF files for more information.

All is not puppies and rainbows, however. While I'm all for making driver installation more predictable, and allowing drivers to be installed offline, there are one or two things about Universal INF files that I find difficult to accept. The most significant of these is that the **ClassInstall32** section is not allowed in Universal INFs. This means that an IHV/OEM cannot install their driver into a unique class that fits their environment. Not being able to use a custom install class also restricts you from defining a default protection on the Device Objects in your devnode. To me, this is not a good thing. I don't want my image scanner or cochlear implant or missile launcher to be lumped into some other random device class. I want my unique devices to show up in their own unique classes. I'm hoping Microsoft re-thinks this restriction, because to me it's just not acceptable.

Oh, you might be thinking, "If I can't use a co-installer when I install my driver, how will I be able to tailor the installation to a particular device/platform/environment/system?" Well, there's an answer for that too. It's called Componentized Installation.

Componentized, Too

Componentized Installation allows you to divide your driver INF file into multiple pieces. You create a Base INF, that supports the basic functioning of your driver. Then you create one or more Extension INFs that can tailor or configure your driver for a particular system make, model, or version.

The key to this is that there's a single INF for all systems that will always get the basic functionality of the device working. The Extension INF(s) can then do any specialized configuration that your driver might require for a particular system or platform. Say, you have special branding that needs to be applied for a given OEM. Or, you want to enable certain value-added functions on particular models of devices that you support. You can accomplish all this from your Extension INF. You write an Extension INF for each unique system type or flavor that you want to support. During the install process, Windows invokes the right INF to do your tailoring. What's nice about this is that the Base INF and the Extension INF can be updated (and distributed via WU) separately. So you, dear driver writer, can get your Base INF working, and your OEM customer(s) can at some later point work through the process of getting their Extension INF set up the way they want.

Componentized Installation and Extension INFs are explained in more detail in the MSDN docs for Extension INFs.

And Then There Are... Hardware Support Apps

Have you heard of Windows 10 S? That's the version of Windows that's "locked down" and only allows you to run apps from the Store. That means that only UWP – not Win32 – apps

(CONTINUED ON PAGE 15)

Universal Drivers... (Cont.)

(CONTINUED FROM PAGE 14)

are supported. Regardless of what you think about this, Microsoft says they're going to sell these systems. And, if I had to guess, there'll be other systems in the future that will have the same restrictions.

Suppose you currently have a Device Manager property sheet, or Control Panel Applet, that's used to manage your device. That's not going to work very well on a system that doesn't support the Win32 API, right? So, the final part of the Universal Driver picture is Hardware Support Apps that are written as UWP apps. Of course, that's not nearly as simple as it might seem at first blush. UWP has some pretty significant limitations: No IOCTLs, no WMI, no RPC. About the only cool technology that you can use in a UWP app is ETW.

In place of device-specific IOCTLs, UWP offers something called Custom Capabilities. These are specific functions your driver authorizes to specific applications. And, in this way, everything is supposed to be kept safe.

I'm not convinced. I'm not a fan of the whole UWP programming model in general (I'm a device guy… I don't like being told what I can and cannot do). But I admit it: I really haven't spent much time investigating the whole issue of Custom Capabilities for UWP apps. Maybe it's not as bad as I think. I'll look into it, and write an article that describes what I find in a future issue of *The NT Insider*. In the meantime, you can read (and weep) more about Custom Capabilities for Hardware Support Apps in MSDN.

Note that there's an interesting side-effect of making a support app a UWP app. That app can now be pulled-down during the install (and also during future updates) from the Windows Store. You can also, optionally, pre-populate the app onto a specific install image using DISM. But in either case, the app will no longer be part of what we think of today as the traditional installer package. This does sound to me like it might make servicing easier.

Gimmie a D! Gimmie a C! Gimmie a H! Gimmie a U!
What's That Spell?
It spells "DCHU." No, you can't pronounce it. It's stands for:

- **Declarative** – You use a Declarative INF file
- **Componentized** – You separate your INF into a Base INF and zero or more Extension INFs
- **Hardware** Support Apps – Written at a UWP app, of course
- **Universal** – Your driver (and INF) is restricted to the Universal feature set

Together, Microsoft describes these as the Universal Driver Pillars. See the lovely artwork I stole from somebody's WinHEC deck in *Figure 3*.

For your driver to be truly able to run on "any Windows platform" you need to embrace all four of the DCHU design principles.

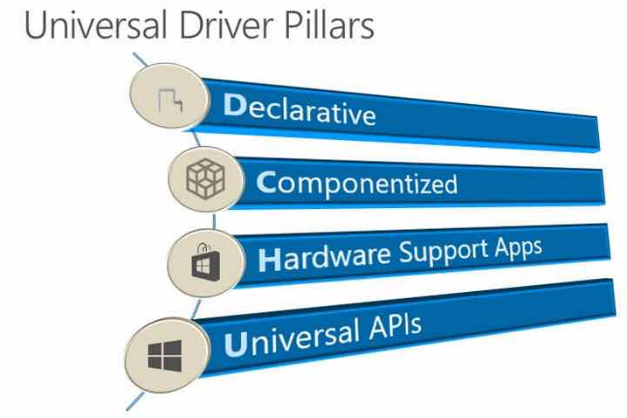


Figure 3 -- The Four Pillars of Universal Drivers
(artwork courtesy of Microsoft from WinHEC 2017)

Universal DDIs – Only Part of the Story
Now, hopefully, you see that writing a driver that's restricted to the Universal DDIs is straight forward, but it's also only part of the story. If you truly want your driver to be installable on any Windows system (now and in the future) you're going to have to become DCHU compliant. Of course, how important any of this is to you will depend greatly on the type of drivers you write and the types of hardware you support.

Be aware that there are rumblings that DCHU compliance will become *required* for certain classes of drivers in the not too distant future. While that remains to be seen, at least the direction that Microsoft *wants* us to move is clear.

File System Minifilters... (Cont.)

(CONTINUED FROM PAGE 9)

Within its PreOperation and/or PostOperation callbacks, a Minifilter can perform almost any operation, including examining or modifying the data involved in the operation. Thus, the trick becomes understanding the specific meaning of those operations.

Win32 API vs Native Windows API

The Filter Manager framework for writing Minifilters is so powerful and well thought-out, that it's very easy for new Minifilter developers to be tricked into thinking that file system filtering is easy. The thing is: It *can* be easy, but it can also be get surprisingly confusing very quickly.

The inherent problem is that regardless of how powerful or pleasant the Filter Manager model is, the world of Windows file systems is inherently a complicated one. Some of this complexity comes from the fact that the Win32 API can sometimes be very different from the native Windows API that the I/O subsystem actually uses. A simple example that we often encounter here at OSR is the Win32 API CopyFile. Developers are often surprised that there's no native analog to this function. Rather, internally, **CopyFile** opens the source file, opens the target file, and if both of those operations are successful, issues a series of reads from the source file and writes to the target file until the file is copied. The source file and the target file are then closed.

A slightly more interesting example is the Win32 function DeleteFile. If you're not a file system dev, you're probably thinking "How complicated can delete be? You just… delete the file, right?" Well, not in Windows, no. The native Windows API doesn't actually have a specific delete file operation. Rather, the intention to delete a file is indicated by issuing a Set Information operation (ZwSetInformationFile) to set the file's Disposition (FILE_DISPOSITION_INFORMATION). This allows the caller to specify whether he wants the file to be deleted when its closed. However, it's important to note that the file is not actually deleted until the last handle to the file has been closed. This means that an application can open a file and call the Win32 **DeleteFile** API, but this doesn't actually delete the file. It just set the file's Disposition of the file to be deleted on close. Even when the app subsequently closes the file, this still doesn't necessarily mean that the file will actually be deleted. Consider what could happen if another application has the file open when our example app opens it. That application can also set the file's Disposition at any time. If after our example app has set the file's Disposition to be delete on close, the other app sets the file's Disposition so that it's no longer marked for delete on close, the file will not be deleted. And yes… it happens.

Virtual Memory System Integration

Another source of complexity for file system Minifilters stems from the close working relationship between file systems, the Windows Cache Manager, and the Windows Memory Manager. We all know that when we call ReadFile, data from the specified data buffer is read from the file indicated by the file handle. Most of us probably also know, at least in a general way, that there's typically some caching involved. That is, every call we make to **ReadFile** doesn't always necessarily result in that data being read directly from the media.

Interestingly, except when a file is explicitly opened with no caching, file systems typically do very little to process an application's call to **ReadFile**, beyond calling the Cache Manager to deal with it. At the file system's request, the Cache Manager copies the data from the cache area associated with that file's open instance into the application's data buffer. This is illustrated in *Figure 2*.

In *Figure 2*, the file system has called the Windows Cache Manager to process an application's **ReadFile** request. The file system identifies the open instance of the file being accessed using the file handle provided by the application in its **ReadFile** call. The file handle "hFile1" is a handle to the indicated File Object which the Cache Manager uses to locate the sections of cached data that are part of the file system's cache.

(CONTINUED ON PAGE 17)

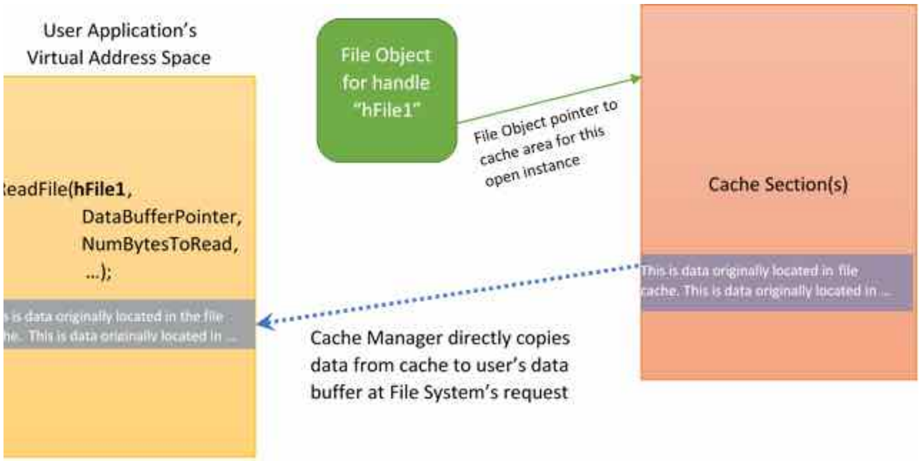

User Application's
Virtual Address Space
File Object
for handle
"hFile1"
File Object pointer to cache area for this open instance
Cache Section(s)
ReadFile(hFile1,
DataBufferPointer,
NumBytesToRead,
…);
This is data originally located in file cache. This is data originally located in …
s is data originally located in the file
he. This is data originally located in …
Cache Manager directly copies data from cache to user's data buffer at File System's request

Figure 2 -- File System Uses Cache Manager to Process a Read Operation

File System Minifilters... (Cont.)

(CONTINUED FROM PAGE 16)

Interestingly, the data stored in cache is read into memory from disk by the Windows Memory Manager via page fault handling. That paging read operation will also pass through the file system.

File system Minifilters can of course be involved in the processing of read and write operations, both originating from applications and originating from the Memory Manager.

Keeping It Simple: Standard Minifilters

The most common type of file system Minifilter monitors, and perhaps tracks or records, various operations performed at the file system level. Some Minifilters, such as antivirus scanners, might even approve or disapprove certain operations. These filters do not, however, become involved in changing the view, or the size, of the data in the files they filter. We term such filters **Standard Minifilters**, because they represent the vast majority of the file system Minifilters that exist.

The primary challenges that developers of Standard Minifilters face are:

A. Properly understanding and dealing with native Windows I/O Subsystem semantics.
B. Properly preserving the behavior of all native file system operations, even the ones they don't necessarily understand or care about.

Of course, this is in addition to the basic challenges inherent in writing any Windows kernel-mode driver.

Overcoming the first of these challenges, that is properly understanding and dealing with native Windows I/O Subsystem semantics, simply requires time and experience. There are numerous Microsoft-provided examples that can be used as a starting point. And the documentation on writing basic file system Minifilters is surprisingly good – Assuming you take the time to read it and understand it. There are also good online resources, like our NTFSD list, that can help by answering questions you might have along the way. And, I should also mention, we'll be teaching a seminar in how to write Standard Minifilters starting in January 2018.

The complexity of item B, above, depends greatly on the type of Minifilter you write. The simpler you keep your operations, and the more limited you keep your filtering, the easier things will be. For example, it's

(CONTINUED ON PAGE 18)

File System Minifilters... (Cont.)

(CONTINUED FROM PAGE 17)

usually a mistake to attempt to filter I/O operations that your filter doesn't specifically care about. The HLK tests for file system Minifilters will be very helpful to you in assessing your success in preserving the behavior of the file systems you filter. Be sure you run them!

Thus, if you need to write a Standard Minifilter – that is, one that monitors file system operations – the challenges you face will be reasonable.

Exponentially Harder: Isolation Minifilters

A much less common type of file system Minifilter is the **Isolation Minifilter**. Isolation filters separate (or "isolate") the **view** of a file's data from the **actual underlying data** stored by the file system. Writing this type of Minifilter is usually as complex as writing a standard Windows file system, because it involves direct and close interaction between your Minifilter, the Windows Cache Manager, and the Windows Memory Manager. In fact, some experienced devs consider Isolation Minifilter even more difficult than file system development, because when you're writing an Isolation Minifilter you effectively have to "fit" the implementation of a Windows file system into the API provided by Filter Manager. Thus, in addition to the challenges faced by the developers of Standard Minifilters, developers of Isolation Minifilters deal with numerous significantly more complex issues.

To illustrate what we mean by "separating the view of a file's data form the actual underlying data", let's consider an Isolation Minifilter that implements on-access transparent encryption/decryption. An application opens, and calls **ReadFile** for, a file that is stored on disk in encrypted form. The encryption/decryption Minifilter's job in this case is to transparently decrypt the data that is presented to the application. The general scheme for supporting this type of activity is shown in *Figure 3*.

Figure 3 shows two File Objects, each associated with the same file. The upper File Object (in green) is the open instance associated with the application. This File Object represents the application's view of file. Note that the cache section referenced by the File Object contains decrypted data. The data was placed into cache by the Isolation Filter, working in cooperation with the Cache Manager.

The lower File Object (in blue) is created by the Isolation Minifilter, and represents the Minifilter's (and the underlying file system's) view of the file. Note that the data in the cache section for this File Object is encrypted. The Isolation Minifilter uses the lower File Object to interact with the underlying file system.

Thus, when the user application calls **ReadFile**, the Isolation Minifilter will receive that request (as part of its read PreOperation processing, discussed previously) and interact with the Cache Manager to fill the cache with data. In processing the subsequent paging read operation (sent by the Memory Manager in fulfillment of the Cache Manager's attempt to fetch data for the cache) the Isolation Minifilter issues a **ReadFile** operation referencing the lower (blue) File Object. The underlying file system, working with the Cache Manager, fulfills this request and puts data into the indicated cache section.

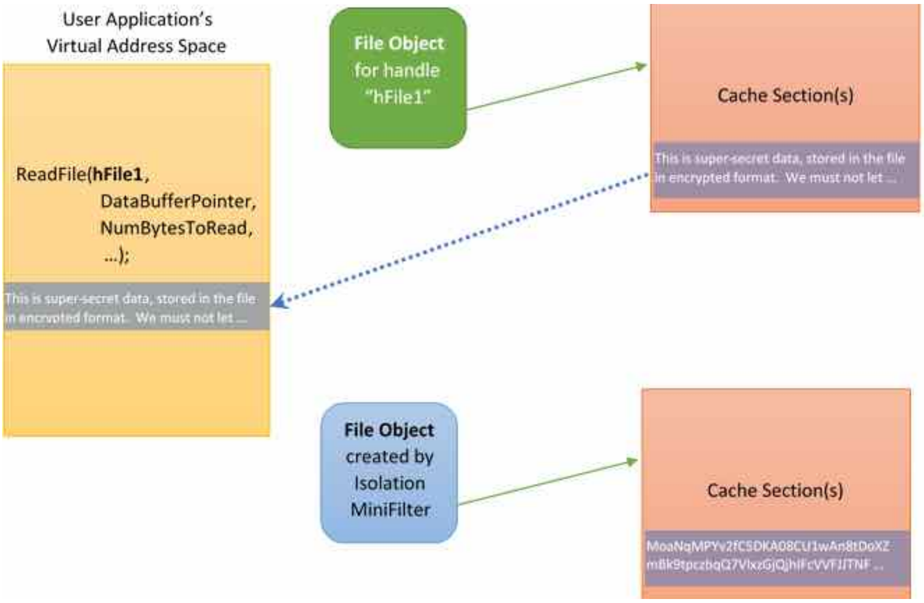

User Application's
Virtual Address Space
ReadFile(hFile1,
DataBufferPointer,
NumBytesToRead,
...);
This is super-secret data, stored in the file in encrypted format. We must not let ...
File Object for handle "hFile1"
Cache Section(s)
This is super-secret data, stored in the file in encrypted format. We must not let ...
File Object created by Isolation MiniFilter
Cache Section(s)
MoaNqMPYv2fC5DKA08CU1wAn8tDoXZ mBk9tpczbqQ7VlxzGjQjhIFcVVFJJTNF ...

Figure 3 -- Isolation Minifilter

Things get even more complex when you consider that most transparent data encryption systems utilize some sort of metadata header (to hold keys or access information). This header is often stored within the file itself. Thus, the file offset used for read operations by the application can be different from the file offset that the Isolation Minifilter needs to use when accessing the file as stored on disk.

But there's usually even more complexity than we've discussed so far. Let's say the encryption/decryption Minifilter determines

(CONTINUED ON PAGE 19)

File System Minifilters... (Cont.)

(CONTINUED FROM PAGE 18)

on a per-application basis whether decrypted data or encrypted data should be provided to the application when it does a read. For example, the Isolation Minifilter might automatically decrypt data from an encrypted document when accessed by Microsoft Word. However, when that same encrypted document is accessed from a backup application, the Isolation Minifilter could provide the raw, encrypted, contents of the file. What's even more interesting is a well-designed Isolation Minifilter could allow both views, the decrypted view of the file's contents and the encrypted view of the file's contents, to different applications reading the file simultaneously. And what happens if one of those applications writes data to the file, while the other has a different view of the file open? Let's just say that things get "interesting," fast.

Isolation Minifilters: Not Just Encryption

We should hasten to add that the Isolation Minifilter model is not only used for transparent data encryption and decryption systems. The Isolation Minifilter model is used any time:

- There's a difference between an application's view of the data and the underlying file stored on the file system.
- There's a difference between the applications view of the data and the underlying location of the file data.
- Different application instances need to having unqiue views of the same underlying data stored on the file system.

For example, consider a change tracking and versioning Isolation Minifilter. One copy of Word might be actively editing the latest version of a file, while a different instance of Word (perhaps being run by a different user) might be actively editing an older version of the file.

Or consider a file that's stored in the cloud and only pulled down bit by bit, based on how the file is accessed. The file might appear to applications to be entirely resident on the local file system. However, an Isolation Filter could be accessing and back-filling data from remote storage as required.

Where Does That Leave Us?

The Windows MiniFilter model is flexible, and extremely powerful. With enough time and experience, most developers will be able to write Standard Minifilters. These filters monitor and/or track file operations, and potentially authorize access to files stored in the underlying file systems. The primary challenges inherent in developing Standard Minifilters are understanding native Windows file system semantics, and transparently passing-through to applications all the functionality the underlying file system offers. Both of these challenges are tractable, and can be accomplished without a vast amount of Windows file system expertise.

In contrast to Standard Minifilters, there are Isolation Minifilters. These Minifilters separate the view an application receives of a file, from the actual underlying data of that same file. Developing an Isolation Minifilter is more like developing a full Windows file system than developing a Standard Minifilter, because it requires close interaction with the Windows Cache Manager and Memory Manager. As such, the development of an Isolation Minifilter is not likely to be a task that can be successfully undertaken by most developers who do not also seek to become Windows file system experts.

Cleanup, Close and Cancel... (Cont.)

(CONTINUED FROM PAGE 7)

In addition to calling *EvtFileCleanup*, WDF goes one step further in its Cleanup processing. After calling the driver's *EvtFileCleanup* Event Processing Callback (if any), WDF scans your driver's WDF Queues for Requests originally submitted via the File Object that's the target of the Cleanup. For each such Request found, WDF issues a Cancel operation. We'll discuss the details of Cancel later, but for now let's just say that issuing a Cancel operation for these Requests almost certainly results in their being removed from the Queue and completed by WDF with an error status.

It is important to note, however, that as part of handling Cleanup, WDF does not perform Cancel processing for pending Requests that are not stored on a WDF Queue. Thus, if you have an in-progress Request, and you're keeping track of that Request by storing its WDFREQUEST handle in your Device Context, that Request will not automatically be Canceled by WDF as part of Cleanup processing and therefore will remain in progress.

There are some other interesting special cases where Windows instantly follows a Cleanup operation with a Cancel for Requests sent via the associated File Object. For a discussion of these cases, see the sidebar **When Cleanup Becomes Cancel** (below).

To summarize: When an application calls **CloseHandle**, your driver will be called at its *EvtFileCleanup* (NOT *EvtFileClose*, as is commonly thought) Event Processing Callback. WDF will then issue a Cancel for any Requests that are resident on a WDF Queue.

When the last reference to a File Object goes away (that is, the File Object's reference count is decremented to zero) Windows issues a Close request and WDF calls the associated driver's *EvtFileClose* Event Processing Callback. The reference count on the File Object being zero means (for the purposes of this article) that there are no I/O operations in progress or pending for the File Object.

If there are no I/O requests outstanding when **CloseHandle** is called, *EvtFileClose* will be called immediately after *EvtFileCleanup*. Your driver's *EvtFileClose* Event Processing Callback will be called at IRQL PASSIVE_LEVEL, and in an arbitrary process and thread context. Thus, when you driver's *EvtFileClose* is called you do not know which thread is "current thread" or which process is "current process".

To summarize: When the File Object reference count goes to zero, signaling that there are no further I/O requests pending for the File Object, your driver will be called at its *EvtFileClose* Event Processing Callback.

(CONTINUED ON PAGE 21)

When Cleanup Becomes Cancel

From reading the article, you already know that when WDF receives a Cleanup operation for a given File Object, it issues a Cancel operation for any Requests issued via the File Object that are on WDF Queues. There are two other times when the Windows I/O subsystem extends Cleanup processing by issuing a Cancel. These are when the I/O operations that are issued are associated with an I/O Completion Port, and when the I/O operations that are issued come from a thread in a Thread Pool.

Completion Ports and Thread Pools are two optimized mechanisms designed to efficiently handle asynchronous callbacks to an application. The case we're particularly interested in is handling of I/O completion callbacks.

When the last handle to a File Object is closed, Windows always issues a Cleanup operation. What makes the processing of **CloseHandle** operations associated with Completion Ports and Thread Pools special is that after the Cleanup operation has completed, the I/O Subsystem issues a Cancel operation for any I/O operations that remain in progress from the File Object. This results (or, rather, "should result" if the relevant drivers are properly doing their jobs) in any outstanding I/O operations that were initiated via the File Object being quickly terminated. Thus, the Completion Port or Thread Pool is promptly notified of the termination of any pending I/Os.

Cleanup, Close and Cancel... (Cont.)

(CONTINUED FROM PAGE 20)

Our final definition is for Cancel. When a thread exits with pending I/O, or an application explicitly calls the Win32 function CancelIo or CancelIoEx, the pending I/O requests are Canceled. This results in WDF calling one of the driver's Cancel-related Event Processing Callbacks, if one has been specified, for each Request to be canceled.

Unlike Cleanup and Close, in which the operation takes place based on a specific File Object, Cancel operations are managed in Windows on an individual, specific, Request basis. A great feature of WDF is that it is designed to automatically assist your driver with Cancel processing. The chief way it does this is that when Windows Cancels a given Request, WDF will search all your driver's WDF Queues for that Request. If it finds the Request that's being Canceled on a WDF Queue, WDF will remove the Request from the Queue and, by default, complete the Request with STATUS_CANCELLED and an information field of zero. This even applies to Requests that are on a Queue waiting to be presented for the first time to your driver (that is, Requests that your driver has not seen yet).

Consider how this helps driver processing in our second example, in which an application exits with multiple Requests in progress. In this case, as the application exits, Windows will issue a Cancel operation for each I/O request that is still in progress when the application attempts to exit. When WDF receives this Cancel operation, it will search your driver's WDF Queues for the Request being Canceled, and if it finds the Request, it will remove it from the Queue and complete it (with STATUS_CANCELLED). If the Canceled Request was waiting on your driver's Queue, and had not yet been presented to your driver for processing, the Request is removed from the Queue and completed without your driver ever having to deal with it. Once again, WDF makes driver writing easier, through its principle of "reasonable defaults."

Of course, you might not want Requests that you're holding on a Queue to be automatically removed from the Queue and completed by WDF during Cancel processing. For example, this might be the case when your driver holds Requests on a Queue with Manual Dispatching while it's waiting for a device to finish its work. In this case, your driver can elect to handle Cancel processing itself for that particular Queue by registering an *EvtIoCanceledOnQueue* Event Processing Callback for the Queue. If your driver has provided an *EvtIoCanceledOnQueue* Event Processing Callback for a Queue, whenever a Request on that Queue is Canceled, WDF will remove the Request from the Queue and call your driver's *EvtIoCanceledOnQueue* Event Processing Callback with the handle to the Request being Canceled.

If your driver holds Requests in-progress, but not on a Queue, it can register an *EvtRequestCancel* Event Processing Callback for a specific Request. For example, if your driver starts a Request on your device and you're waiting for the device to complete its processing, your driver might simply save the handle to the pending Request in a location in your Device Context. This is, in fact, very common. If you want to be able to handle Cancel events for that Request while it is pending, waiting for your device, you can optionally register an *EvtRequestCancel* Event Processing Callback for that Request.

To Summarize: Whenever an application begins its exit processing with I/O in progress, or an application calls CancelIo or CancelIoEx, your driver may be called at its *EvtRequestCancel* or *EvtIoCancledOnQueue* Event Processing Callback.

A handy summary of Cleanup, Close, and Cancel processing is shown in table form in *Figure 2* (next page).

Given the above definitions, let's examine how your driver can use the Cleanup, Close and Cancel events to its best advantage.

(CONTINUED ON PAGE 22)

Cleanup, Close and Cancel... (Cont.)

(CONTINUED FROM PAGE 21)

Event	Resulting Operation	Callback Constraints
Last handle to a File Object closed	CLEANUP: *EvtFileCleanup* invoked; WDF issues a Cancel operation for any Request that is currently on any WDF Queue.	IRQL PASSIVE_LEVEL, in the context of the thread that called **CloseHandle**
Last reference to a File Object removed	CLOSE: *EvtFileClose* invoked	IRQL PASSIVE_LEVEL, in an arbitrary process/thread context
Application begins exit process, Application calls CancelIo or CancelIoEx	CANCEL: Request-dependent cancel function (either *EvtRequestCancel* or *EvtIoCanceledOnQueue*)	IRQL <= DISPATCH_LEVEL, in an arbitrary process/thread context

Figure 2-- Summary of Cleanup, Close, and Cancel

When Should You Handle Cleanup?

When WDF gets a Cleanup notification for a particular File Object, it calls your driver's *EvtFileCleanup* Event Processing Callback if one has been specified. *EvtFileCleanup* has the following prototype:

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

Should your WDF driver implement Cleanup? If so, what should you do as part of Cleanup processing?

The answer is simple for most drivers: In almost all cases you do not need to handle Cleanup – You therefore should not provide an *EvtFileCleanup* Event Processing Callback in your driver.

Cleanup operations in Windows are designed for basic two purposes:

1) As a quick notification to the driver that the calling application does not intend to issue any further I/O requests on the associated handle (remember, Cleanup happens on a per File Object basis);
2) To provide the driver a "last chance" to perform any operations that need to be done in the context of the requesting application (as the application is indicating its intention to stop interacting with the driver via this specific open instance).

As part of item 1, above, Cleanup can be used as a wholesale method of discarding a bunch of Requests that probably no longer matter to the application that issued them. WDF facilitates this by performing a Cancel operation on any Requests that were issued on the File Handle being closed and that are on a WDF Queue when the Cleanup is processed. The primary result of this is that any Requests from the Cleaned-up File Object that were on a Queue awaiting presentation to your driver will never reach your driver. WDF will remove them from the Queue and Cancel them, without your driver having to do anything.

Item number 2, above, is slightly more interesting. Because *EvtFileCleanup* is called in the context of the process/thread that called **CloseHandle**, this Callback gives your driver the opportunity to tear down any context that it was maintaining on a per-process basis. For example, if your driver mapped some memory into the application's virtual address space, the *EvtFileCleanup* Event Processing Callback is the perfect place to undo that mapping. This type of operation isn't common. But it is one of the key uses of Cleanup in WDF. And, yes, before the expert readers raise an uproar, let me add that there's more to mapping and unmapping memory into an application's virtual address space than simply unmapping in *EvtFileCleanup*. I'm not attempting to address that here.

It's important to note that there's also a nasty little detail inherent in Cleanup handling: Since handles are per-process entities, one thread in a process can be sending I/O requests on a handle (repeatedly calling **ReadFile**, for example), while another thread simultaneously calls **CloseHandle** on that same handle. If you think about this, you'll realize that this means that it's possible for

(CONTINUED ON PAGE 23)

Cleanup, Close and Cancel... (Cont.)

(CONTINUED FROM PAGE 22)

your driver to receive one or more new Requests on a given handle, even after your driver has been called at its *EvtFileCleanup* Event Processing Callback for that handle. This makes for a nasty race condition, and contributes to making Cleanup the wrong place to definitively purge Requests from your driver to avoid, for example, an application hanging on exit.

Thus, in general, WDF drivers do not need to – and in fact should not – directly handle Cleanup. As I always say "If you don't write any code, you don't have any bugs." So, when you don't need to implement a function, that's usually a good thing.

When Should You Implement Close Handling?

When WDF gets a Close notification for a particular File Object, it calls your driver's *EvtFileClose* Event Processing Callback if one has been specified. *EvtFileClose* has the following prototype:

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

Should your WDF driver implement Close? If so, what should you do as part of Close processing?

As with Cleanup, for most drivers the answer is simple: For most drivers, there's no need to handle Close, and you should therefore not provide an *EvtFileClose* Event Processing Callback in your driver.

Close is your last notification that a File Object is going away. When *EvtFileClose* is called, there are no more Request pending on the File Object. As soon as you return from your *EvtFileClose* routine (if you provide one) the associated File Object will be destroyed.

So, what's the purpose of Close? The primary purpose of Close is to allow your driver a "last chance" to tear down any handle-related context it might have been maintaining. For example, suppose that during Open processing (called at **CreateFile** time) your driver allocates some data structures that it uses for tracking various state and information. Or maybe your driver tracks the number of open instances of its devices. In both these cases, *EvtFileClose* provides you the perfect opportunity to reverse those operations. To return the handle-related context, for example. Or to decrement the count of open instances.

So, in general, WDF drivers do not need to – and in fact should not – directly handle Close. The only time you need to do so is when you have per-handle context that you need to tear-down. In that case, *EvtFileClose* is the right place to handle this tear-down.

How Should You Deal with Cancel?

Remember that, unlike Cleanup and Close, Cancel notification is implemented on a per-Request basis. As we said earlier, when a thread exits with I/O in progress, or an application explicitly requests one or more I/O operations to be terminated before they are completed, Windows issues a Cancel for specific Requests, one at a time.

The purpose of Cancel is to tell a driver that it should stop processing a given I/O operation as soon as it is convenient for the driver to do so, because the results of that I/O operation are no longer relevant. As we described earlier, the most common use of Cancel is as part of thread and process exit. When a thread exits, Windows will Cancel any uncompleted I/O requests from that thread. This is referred to as I/O rundown processing. I/O

(CONTINUED ON PAGE 24)

Cleanup, Close and Cancel... (Cont.)

(CONTINUED FROM PAGE 23)

rundown is important because, as we noted earlier, Windows will not allow the process to exit completely until its I/O operations are completed. This prevents a driver from unintentionally corrupting memory merely by processing an otherwise valid Request.

Should your WDF driver directly handle Cancel? And, if so, how should it handle it?

In general, if your driver does any asynchronous Request processing – that is, it does not immediately complete every Request it receives within its associated *EvtIo* Event Processing Callback – it should implement Cancel. If your driver leaves one or more Requests in-progress, such as while waiting for a device to perform a particular function, your driver must implement Cancel.

The reason for these rules should be clear: Because Windows will not allow a process to exit with in-progress I/O, the process will be made to wait for any I/O requests to complete before it can exit. If you don't implement Cancel, the process will hang until all its I/O operations have completed in their ordinary course way. If you're absolutely certain that every Request you start in your driver will complete quickly, you're probably OK not handling Cancel (we'll talk more about this in the next section). On the other hand, if a Request can remain in progress for a long, or even an unknown, amount of time, you must implement Cancel to terminate that Request and allow the issuing thread/process to exit.

WDF provides some help by automatically handling Cancel for Requests that happen to be on a WDF Queue when they are Canceled. WDF removes the Request from the Queue and calls your driver's *EvtIoCanceledOnQueue* Event Processing Callback associated with the Queue if one has been supplied. The prototype for this function is:

```
VOID EvtIoCanceledOnQueue(
  _In_ WDFQUEUE   Queue,
  _In_ WDFREQUEST Request)
```

When this Callback is called, the indicated Request has already been removed from the indicated Queue. How your driver deals with this Request is up to you. However, you cannot return the Request to a Queue – you must complete it, either within your *EvtIoCanceledOnQueue* Event Processing Callback or later. The general intention of

(CONTINUED ON PAGE 25)

Cleanup, Close and Cancel... (Cont.)

(CONTINUED FROM PAGE 24)

EvtIoCanceledOnQueue is to allow your driver to stop whatever hardware operation that may be in progress as a result of the Request, and then have your driver complete the Request with STATUS_CANCELLED and an information field of zero.

If a *EvtIoCanceledOnQueue* Event Processing Callback has not been supplied, WDF simply removes the request from the Queue and completes it with STATUS_CANCELLED and an information field of zero. As we also described earlier, one of the reasons this is so helpful is that it automatically dispenses with Requests that are Queued but not yet presented to your driver.

If your driver holds one or more Requests in progress, but not on a Queue, then your driver will need to specify a *EvtRequestCancel* Event Processing Callback for each of these Requests. Your driver does this by calling **WdfRequestMarkCancelable**, specifying a Request handle and pointer to an *EvtRequestCancel* Event Processing Callback implemented by your driver. The *EvtRequestCancel* Callback has the following prototype:

```
VOID EvtRequestCancel(
  _In_ WDFREQUEST Request)
```

Again, the goal for your driver in this Callback is to stop any device operations associated with the Request that happen to be in progress, and complete the Request with STATUS_CANCELLED and an information field of zero.

Comments on Cancel: The Real World

While I've tried to make the general goals and processes for handling Cancel in your driver clear, I hope it's also evident that handling Cancel can sometimes be difficult. And, sometimes, it can be very, very, difficult. For instance, it's often not exactly easy to stop a device that's in the midst of processing an I/O operation. Or, worse, it's often not easy to stop a device from processing one single, specific, I/O operation when it's in the midst of processing several (hundred or thousand) I/O operations simultaneously.

Thus, as alluded to earlier, in some cases it's probably best to not implement any Cancel handling at all for in-progress I/O requests. Be aware that this is only a reasonable strategy when you're certain that all your in-progress I/O requests will complete in a timely manner. Yes, when you're notified of a Cancel operation there's a thread and/or process that's waiting to exit. But if it takes another 10ms or 20ms to complete a DMA transfer, and then complete the pending Request normally, that amount of time delay isn't going to matter to the thread or process. And that's a whole lot easier than trying to cancel the in-progress operation. Trust me. I have the scars to prove this.

There's also another detail you need to know: Unless you're doing so from within your *EvtRequestCancel* Event Processing Callback, you must never call **WdfRequestComplete** (or any variant therefore) with a Request that's been marked as Cancelable. This means that if you associate a Cancel routine with a Request, and you complete that Request as part of your regular processing (that is, you don't complete it as part of Cancel processing) you must first call **WdfRequestUnmarkCancelable**.

Unfortunately, Cancel handling remains one of the most complex parts of I/O operations. While WDF provides you with some help by canceling Request that reside on a Queue and have yet to be presented to your driver, it doesn't do much to help you with Requests that have been presented to your driver and are in progress when they're canceled.

Back to Those Examples

To summarize what we've discussed, let's return again to the examples we used at the start of this article. In our first example, an application sends 100 asynchronous Read Requests to your driver, and then calls **CloseHandle**, while some of those Request are still in progress. Your driver uses a single Queue with Sequential Dispatching to service these Requests.

Now, you know what happens.

As soon as the application calls **CloseHandle**, Windows will immediately issue a Cleanup. If your driver needed to handle any state specific to the process that was doing the **CloseHandle** operation, it would provide an *EvtFileCleanup* Event Processing Callback. From within this Callback, it would tear down the process specific state. Whether or not your driver provided an *EvtFileCleanup* Event Processing Callback, WDF will Cancel any Requests that are pending on your driver's Queue.

(CONTINUED ON PAGE 26)

Cleanup, Close and Cancel... (Cont.)

(CONTINUED FROM PAGE 25)

Assuming your driver does not provide a *EvtIoCanceledOnQueue* Event Processing Callback for your top-edge Queue (a safe assumption), WDF will remove from the Queue and terminate any pending Requests that were issued on the handle. Thus, the vast majority of Requests that were in progress will be terminated automatically by WDF.

If you driver has a Request (and, it can only be one… the example uses Sequential Dispatching) in progress when the application calls **CloseHandle**, unless that Request is on a Queue when the Cleanup is received, that Request will remain in progress in your driver. Either your driver will complete it as part of its ordinary processing, or the Request will be canceled when the issuing thread exits. Either way, there's no problem. And, whenever the last I/O operation is completed on the File Object, your driver's *EvtFileClose* Event Processing Callback will be called, if you specified one.

So, that's what happens in our first example.

In our second example, the application issued the same 100 reads, but instead of closing the handle it exited or was terminated.

In this case, Windows first undertakes I/O rundown processing for the indicated thread. Windows will issue a Cancel operation for each of the in-progress Requests. Thus, as described previously, assuming you have not specified an *EvtIoCanceledOnQueue* Event Processing Callback for your top-edge, WDF will remove from the Queue and terminate each Canceled Request.

If your driver is holding a Request in progress, and the Request is not guaranteed to complete quickly, you will have specified a per-request *EvtRequestCancel* Event Processing Callback by calling **WdfRequestMarkCancelable**. Within that callback, your driver will complete the Request with STATUS_CANCELLED.

Of course, if your driver is holding a Request in progress, and the Request is absolutely guaranteed to complete quickly, your driver need do nothing more.

Thus, at the end of Cancel handling or very shortly thereafter, there will be no more I/O Requests in progress from the application to your driver.

When all the pending Requests have been completed, Windows will issue a Cleanup (and WDF will call your driver's *EvtFileCleanup* Event Processing Callback if one has been specified) and then a Close (and WDF will call your driver's *EvtFIleClose* Event Processing Callback if one has been specified). Unless your driver needs to tear-down some per-process specific state, it won't have specified a Cleanup Callback. And unless your driver needs to tear-down per-handle specific state, it won't have provided a Close Callback.

Now You Know

Now you know the meaning of Cleanup, Close, and Cancel in Windows. You know how WDF helps you handle these operations. And you know, at least in general terms, what your driver needs to do to handle each of them.

Hopefully you'll never again be confused by these three similar sounding operations!

FESF for Linux... (Cont.)

(CONTINUED FROM PAGE 3)

Compatibility with Windows as a Bonus

FESF for Linux is useful on its own, as the basis for a purely Linux product. However, it also has the advantage of being compatible with Windows.

Because the On Disk Structure used by FESF for Linux is compatible with the one used by FESF for Windows, encrypted files are fully compatible between systems. Now you can create an encrypted document using Microsoft Word, and have it transparently decrypted on access, based on any policy you define. Make changes using LibreOffice Writer, save the document (transparently encrypted, of course), and send it back to your Windows colleagues. They can then open it with Word, where it will be automatically and transparently decrypted (assuming your policy allows this).

Early Adopter Program Open NOW

We've already started work on FESF for Linux. You can join our Early Adopter Program to influence the direction of the first release of the product, get early access to pre-release builds, and help us by providing important feedback as development progresses. For more information about FESF for Linux see our web page. For more information about our FESF for Linux Early Adopter Program contact sales@osr.com.

OSR Seminar Schedule

Seminar	Dates	Location
Internals & Software Drivers	13-17 November	Dulles/Sterling, VA
Kernel Debugging & Crash Analysis	4-8 December	**At OSR!** Amherst/Nashua, NH
WDF Drivers I: Core Concepts	8-12 January	**At OSR!** Amherst/Nashua, NH
WDF Drivers II: Advanced	15-18 January	**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.